Berechenbarkeit und Komplexität

eine Mitschrift zur Vorlesung von

Prof. Dr. Wolfgang Thomas

im Wintersemester 1998/99

vorläufige Version vom

26. Juli 1999

Dieses Skript wurde erstellt von:
Matthias Egerland
Florian Hasibether
Simon Kirstein
Wolfgang Thomas

Inhaltsverzeichnis

Einleitung

Aufgabe der theoretischen Informatik ist die Analyse der in der Informatik auftretenden Probleme und Strukturen mit mathematischen Methoden. Im Zentrum stehen dabei naturgemäß Begriffe wie „Algorithmus“, „Programm“, „Programmiersprache“, „Rechnung“ und „Effizienz“. Insbesondere wird versucht, diese und ähnliche Begriffe präzise zu definieren, ihre Mächtigkeit (Tragweite und Grenzen) zu erfassen und Anwendungen bzw. Weiterentwicklungen zu finden.
Etwas konkreter gefaßt, geht es um Fragen wie diese: Wie kann man Algorithmen auffinden, notieren und auf ihre „Leistung“ (Korrektheit, Effizienz) untersuchen bzw. diese Leistung sicherstellen? Gibt es algorithmische Probleme, die sich nicht algorithmisch lösen lassen? (Das ist die Frage nach absoluten Gesetzen und Grenzen, so wie die Lichtgeschwindigkeit eine prinzipielle Grenze in der Physik darstellt.) In welchem Ausmaß läßt sich der algorithmische Aufwand zur Lösung eines Problems minimieren? (Bei kryptographischen Verfahren will man umgekehrt Probleme benutzen, die unüberwindliche Hürden für die algorithmische Lösung enthalten.)
Die Methoden und Ergebnisse der theoretischen Informatik haben zu einer Präzisierung dieser Fragen und zu recht weitgehenden Antworten geführt. In dieser Vorlesung werden die genannten Grundbegriffe der Informatik (Algorithmus, Programm, Programmier-Sprache, Rechnung, Effizienz) präzisiert bzw. anhand von Muttersprachen eingeführt. Unser Ziel ist die Klärung folgender Fragen: Was sind die Grenzen

- für die Lösung algorithmischer Probleme,
- für die effiziente Lösung algorithmischer Probleme.

Hierzu gehen wir in drei Etappen vor:

1. Präzisierung der Begriffe „algorithmisches Problem“, „Lösung“, „effiziente Lösung“,
2. mathematische Analyse der präzisen Begriffe,
3. Rückschlüsse auf Praxis der Informatik.

Kapitel 1

Algorithmenbegriff und Turingmaschinen

1.1 Wörter, Zahlen und Funktionen

Algorithmen (Programme, Informatik-Systeme) transformieren *Daten*, die sie von ihrer Umgebung erhalten.

In dieser Vorlesung betrachten wir „digitale Verfahren“, die

1. schrittweise (getaktet) arbeiten,
2. Daten manipulieren, die letztlich durch Bitwörter (endliche 0-1-Folgen) dargestellt werden können.

Da der Bezug auf Bitwörter für viele Anwendungen umständlich ist, lassen wir *Wörter aus Zeichen eines endlichen Symbolvorrats* (z.B. Tastatur) zu.
In weiten Bereichen der Informatik, insbesondere für Grundlagenfragen, sind Wörter, Wortmengen und Wortfunktionen die fundamentalen Objekte der Untersuchung. Der Grund hierfür liegt in der Fokussierung der Untersuchungen auf Algorithmen bzw. Programme und die durch sie manipulierten Daten. Sowohl Programme als auch die Daten können in natürlicher Weise als Zeichenreihen (Wörter) dargestellt werden, etwa wenn sie auf einer Tastatur Zeichen für Zeichen eingegeben werden. Ganz entsprechend läßt sich eine Programmiersprache als die Menge der zulässigen Programme, also als eine Wortmenge ansehen. Eine Programmübersetzung, z.B. in Maschinencode, realisiert in dieser Sicht eine Abbildung von Wörtern auf Wörter, also eine Wortfunktion. Auch die Ausführung eines beliebigen Programms stellt dann eine Abbildung von Eingaben auf Ausgaben dar und ist somit durch eine Wortfunktion beschreibbar. Schließlich läßt sich in vielen Fällen auch die Programmausführung, d.h. eine „Rechnung“ in der getakteten Zeit, durch eine (endliche oder unendliche) Folge von Zeichen für die „Einzelsituationen“ repräsentieren, wodurch die Menge aller möglichen Rechnungen wieder durch eine formale Sprache erfaßt wird.

Ein *Alphabet* ist eine endliche, nichtleere Menge Σ von Zeichen (Symbolen).

Beispiel:
$$\begin{array}{lcl} \Sigma_{bool} & = & \{0,1\} \\ \Sigma_{lat} & = & \{a,b,\ldots,z\} \\ \Sigma_{\text{tastatur}} & = & \Sigma_{lat} \cup \{A,\ldots,Z,\sqcup,:,;,\ldots,!\} \\ \Sigma_m & = & \{\underline{1},\ldots,\underline{m}\} \qquad (\text{für } m \geq 1) \end{array}$$

Buchstaben werden angedeutet durch $a, b, a_0, a_1, \ldots$

Ein *Wort* über dem Alphabet Σ ist eine endliche Folge von Buchstaben aus Σ.
Wörter werden angedeutet durch $u, v, w, u_0, u_1, \ldots$

ϵ bezeichne das *leere Wort* (die leere Buchstabenfolge).
Beachte: Über Σ_{tastatur} ist das Leer-Symbol ein Wort der Länge 1, also $\sqcup \neq \epsilon$.

Die *Länge* $|w|$ eines Wortes w ist die Anzahl der Vorkommen von Buchstaben in w.

Beispiel: (Σ_{tastatur})
$|papa| = 4 \quad |app| = 3 \quad |\epsilon| = 0$

Σ^* sei die Menge aller Wörter über Σ, und $\Sigma^+ := \Sigma^* \backslash \{\epsilon\}$.

Zwei *Operationen* auf Wörtern:

Die *Verkettung* $\cdot : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ ist für $u = a_1 \ldots a_k$, $v = b_1 \ldots b_l$ definiert durch $u \cdot v = a_1 \ldots a_k b_1 \ldots b_l$

Bemerkung 1.1.1 *Es gelten die Gleichungen* $u \cdot (v \cdot w) = (u \cdot v) \cdot w$, $u \cdot \epsilon = \epsilon \cdot u = u$.
Also gilt: $(\Sigma^*, \cdot, \epsilon)$ *ist Monoid.*

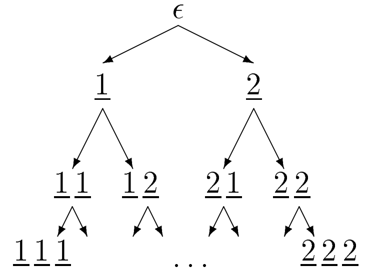
Die *Spiegelbild*(Reversal)-Funktion ${}^R : \Sigma^* \rightarrow \Sigma^*$ ist für $u = a_1 \ldots a_k$ definiert durch $u^R = a_k \ldots a_1$

Bemerkung 1.1.2 *Es gilt:* $\epsilon^R = \epsilon$, $a^R = a$, $(u^R)^R = u$

Wörter und natürliche Zahlen

Wir stellen nun einen Zusammenhang zwischen der Menge Σ^* aller Wörter über dem Alphabet Σ und der Menge $\mathbb{N}$ der natürlichen Zahlen her. Hat Σ genau m Buchstaben, so arbeiten wir mit $\Sigma_{(m)} = \{\underline{1}, \ldots, \underline{m}\}$ als typischem Beispiel.
Die kanonische Reihenfolge der Wörter über $\Sigma_{(m)}$ lautet:
$\epsilon, \underline{1}, \underline{2}, \ldots, \underline{m}, \underline{1}\,\underline{1}, \underline{1}\,\underline{2}, \ldots, \underline{1}\,\underline{m}, \underline{2}\,\underline{1}, \ldots, \underline{2}\,\underline{m}, \underline{1}\,\underline{1}\,\underline{1}, \ldots$
Man geht also der Länge nach vor, und bei fester Länge nach lexikographischer Reihenfolge.
Eine anschauliche Sicht ergibt sich, wenn man die Menge $\Sigma^*_{(m)}$ in Baumform darstellt:

Illustration für $m = 2$



Die kanonische Reihenfolge listet die Wörter aus diesem Baum Level für Level (jeweils von links nach rechts) auf.

Anzahl der Wörter über $\Sigma_{(2)}$ der Länge	l: 2^l,	allgemein über $\Sigma_{(m)}$:	m^l
Anzahl der Wörter über $\Sigma_{(2)}$ der Länge	$\leq l$: $2^{l+1}-1$,	allgemein über $\Sigma_{(m)}$:	$\frac{m^{l+1}-1}{m-1}$.

Der Zusammenhang zwischen Zahlen und Wörtern wird vermittelt durch folgende Funktion: $\delta_m : \mathbb{N} \to \Sigma^*_{(m)}$ sei definiert durch

$$\delta_m(n) = \text{das } n\text{-te Wort in kanonischer Reihenfolge über } \Sigma_{(m)}.$$

Beispiel: $\delta_2(4) = \underline{1}\,\underline{2}$, $\quad \delta_2(7) = \underline{1}\,\underline{1}\,\underline{1}$, $\quad \delta_2(0) = \epsilon$.

Bemerkung 1.1.3 *δ_m ist Bijektion von $\mathbb{N}$ auf $\Sigma^*_{(m)}$.*

Wir können also die Umkehrfunktion $\gamma_m : \Sigma^*_{(m)} \to \mathbb{N}$ definieren durch

$$\gamma_m(w) = \text{das } n \text{ mit } \delta_m(n) = w.$$

$$\mathbb{N} \underset{\gamma_m}{\overset{\delta_m}{\rightleftarrows}} \Sigma^*_{(m)}$$

Wertetabelle für δ_2

$$\begin{array}{rcrcr}
\epsilon & \mapsto & 0 & & \\
\underline{1} & \mapsto & 1 & = & 1 \cdot 2^0 \\
\underline{2} & \mapsto & 2 & = & 2 \cdot 2^0 \\
\underline{1}\,\underline{1} & \mapsto & 3 & = & 1 \cdot 2^1 + 1 \cdot 2^0 \\
\underline{1}\,\underline{2} & \mapsto & 4 & = & 1 \cdot 2^1 + 2 \cdot 2^0 \\
\underline{2}\,\underline{1} & \mapsto & 5 & = & 2 \cdot 2^1 + 1 \cdot 2^0 \\
\underline{2}\,\underline{2} & \mapsto & 6 & = & 2 \cdot 2^1 + 2 \cdot 2^0 \\
\underline{1}\,\underline{1}\,\underline{1} & \mapsto & 7 & = & 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
& \vdots & & & \vdots \\
\underline{k_r} \dots \underline{k_1} & \mapsto & & & k_r \cdot 2^r + k_{r-1} \cdot 2^{r-1} + \dots + k_0 \cdot 2^0
\end{array}$$

Allgemein erhalten wir:

Bemerkung 1.1.4 (Berechnung von δ_m, γ_m)

a) Für $\underline{k_r} \dots \underline{k_0} \in \Sigma^+_{(m)}$ ($k_i \in \{1, \dots, m\}$) gilt $\delta_m(\underline{k_r} \dots \underline{k_0}) = \sum_{i=0}^{r} k_i \cdot m^i$.

b) Für $n > 0$ ist $\gamma_m(n)$ dasjenige Wort $\underline{k_r} \dots \underline{k_0}$ mit $n = \sum_{i=0}^{r} k_i \cdot m^i$.

Wir erläutern die Ermittlung von $\gamma_m(n)$; sie erfolgt durch iterierte Division durch m mit positivem Rest $(1, \dots, m)$.

Beispiele:

$\delta_4(44)$

$$\begin{array}{rcr} 44 & = & 10 \cdot 4 + 4 \\ & \swarrow & \\ 10 & = & 2 \cdot 4 + 2 \\ & \swarrow & \\ 2 & = & 0 \cdot 4 + 2 \end{array}$$

$\delta_4(44) = \underline{2}\,\underline{2}\,\underline{4}$

$\delta_4(99)$

$$\begin{array}{rcr} 99 & = & 24 \cdot 4 + 3 \\ & \swarrow & \\ 24 & = & 5 \cdot 4 + 4 \\ & \swarrow & \\ 5 & = & 1 \cdot 4 + 1 \\ & \swarrow & \\ 1 & = & 0 \cdot 4 + 1 \end{array}$$

$\delta_4(99) = \underline{1}\,\underline{1}\,\underline{4}\,\underline{3}$

Um dieses Verfahren zu rechtfertigen, benutzen wir (ohne Beweis):

Lemma 1.1.1 *Sei $m > 1$, $n \geq 1$. Dann existieren eindeutig bestimmte q, r mit $n = q \cdot m + k$, wobei $q < n$, $1 \leq k \leq m$.*

Wiederholte Anwendung des Lemmas liefert für gegebenes n das folgende Zahlenschema:

$$\begin{array}{rcrcl} n & = & q_0 \cdot m & + & k_0 \\ & \swarrow & & & \\ q_0 & = & q_1 \cdot m & + & k_1 \\ & \vdots & & & \\ & \swarrow & & & \\ q_{r-2} & = & q_{r-1} \cdot m & + & k_{r-1} \\ & \swarrow & & & \\ q_{r-1} & = & 0 \cdot m & + & k_r \end{array}$$

Behauptung: In diesem Fall ist $\delta_m(n) = \underline{k_r} \ldots \underline{k_0}$.

Wegen 1.1.4b genügt es zu zeigen: $n = \sum_{i=0}^{r} k_i \cdot m^i$. Es gilt (mit wiederholtem Einsetzen gemäß Schema):

$$\begin{aligned} n &= q_0 \cdot m + k_0 \\ &= (q_1 \cdot m + k_1) \cdot m + k_0 = q_1 \cdot m^2 + k_1 \cdot m + k_0 \\ &= (q_2 \cdot m + k_2) \cdot m^2 + k_1 \cdot m + k_0 = q_2 \cdot m^3 + k_2 \cdot m^2 + \ldots + k_0 \\ &\vdots \\ &= q_{r-1} \cdot m^r + k_{r-1} \cdot m^{r-1} + \ldots + k_0 \\ &= k_r \cdot m^r + k_{r-1} \cdot m^{r-1} + \ldots + k_0. \end{aligned}$$

□

Nachdem wir Wörter und Zahlen als typische Daten eingeführt haben, wenden wir uns *Datentransformationen* zu. Ein Algorithmus bewirkt eine solche Datentransformation. Betrachten wir Wörter als Daten, handelt es sich um eine *Wortfunktion*.
Wortfunktionen erfassen also das Verhalten von Algorithmen (Eingabe-Ausgabe-Beziehung).

Wir benötigen etwas **Terminologie zu Funktionen**:
$f, g, h, \ldots$ bezeichnen Funktionen.
$\mathrm{Def}(f)$: Definitionsbereich von f
$\mathrm{Bild}(f)$: Bildbereich von f

f heißt *partielle Funktion* von A nach B, geschrieben $f : A- \to B$, falls $\mathrm{Def}(f) \subseteq A$ und $\mathrm{Bild}(f) \subseteq B$.
f heißt *total*, falls $\mathrm{Def}(f) = A$, geschrieben $f : A \to B$.

Falls $a \in A \backslash \mathrm{Def}(f)$, sagen wir „$f(a)$ ist undefiniert“, kurz $f(a) = \perp$ („bottom“).

Wir arbeiten zumeist mit $A = \underbrace{\Sigma_1^* \times \ldots \times \Sigma_1^*}_{n\text{-mal}} = (\Sigma_1^*)^n$, $B = \Sigma_2^*$.

D.h. wir betrachten Algorithmen mit Eingaben, die n-Tupel von Wörtern über Σ_1 sind, und Ausgaben, die Wörter über Σ_2 sind. Ihr Verhalten wird beschrieben durch Funktionen

$$f : (\Sigma_1^*)^n - \to \Sigma_2^*.$$

Übungen

Übung 1 Wir betrachten das Alphabet $\Sigma_3 = \{\underline{1}, \underline{2}, \underline{3}\}$.

(a) Bestimmen Sie die Wörter $\delta_3(10)$ und $\delta_3(100)$.

(b) Bestimmen Sie die Werte $\gamma_3(\underline{2}\,\underline{2}\,\underline{2})$ und $\gamma_3(\underline{3}\,\underline{3}\,\underline{1}\,\underline{1}\,\underline{2})$.

Übung 2 Die lexikographische Ordnung $<_{\mathrm{lex}}$ auf den nichtleeren Wörtern über einem geordneten Alphabet geht wie in einem Lexikon vor. Also gilt (mit der üblichen Reihenfolge der lateinischen Buchstaben) zum Beispiel:

$$papa <_{\mathrm{lex}} papagei <_{\mathrm{lex}} papua <_{\mathrm{lex}} zoo.$$

(a) Geben Sie eine präzise Definition. Sei hierzu Σ ein Alphabet und $<$ eine Ordnung auf Σ. Es ist festzulegen, wann für zwei Wörter $a_1 \ldots a_k, b_1 \ldots b_l$ (mit $k, l > 0$) über Σ

$$a_1 \ldots a_k <_{\mathrm{lex}} b_1 \ldots b_l$$

gilt (mit Rückgriff auf die Ordnung $<$ auf Σ).

(b) Sei Σ ein geordnetes Alphabet mit wenigstens zwei Elementen. Geben Sie eine Wortmenge $W \subseteq \Sigma^*$ so an, dass die Ordnung $<_{\mathrm{lex}}$ auf W *dicht* ist, d.h. dass zwischen je zwei verschiedenen nichtleeren Wörtern von W ein weiteres Wort von W liegt.

(c) Geben Sie über Σ_2 eine unendliche absteigende Kette

$$w_0 >_{\mathrm{lex}} w_1 >_{\mathrm{lex}} w_2 >_{\mathrm{lex}} \ldots$$

an.

Übung 3 Wir betrachten die Übersetzung der üblichen m-adischen Darstellung natürlicher Zahlen (ohne führende Nullen) in die m-adische Darstellung mit positiven Resten. Beispiel für Basis 10: die Zeichenfolge 389001 wird übersetzt in $\underline{3}\,\underline{8}\,\underline{8}\,\underline{9}\,\underline{10}\,\underline{1}$; die Zeichenfolge 0 wird übersetzt in das leere Wort.
Beschreiben Sie ein Verfahren (z.B. durch ein Flussdiagramm), das diese Übersetzung buchstabenweise von der niedrigsten Stelle aus realisiert. In der Eingabe seien (außer im Sonderfall der Null) keine führenden Nullen erlaubt, und vor der ersten Ziffer stehe ein Anfangsmarker $\cent$.

Übung 4 Gegenstand dieser Aufgabe ist der Übergang von einem beliebigen Alphabet zum zweielementigen Alphabet Σ_{bool}.
Sei $m \geq 1$. Eine *Kodierung der Wörter über Σ_m durch Wörter über Σ_{bool}* ist eine Funktion $f : \Sigma_m \to \Sigma^*_{\text{bool}}$ derart, dass für alle $a_1, \ldots, a_n, b_1, \ldots, b_k \in \Sigma_m$ $(k, n \geq 0)$ gilt: Wenn $f(a_1) \ldots f(a_n) = f(b_1) \ldots f(b_k)$, so $k = n$ und $a_i = b_i$ für alle $i \leq n$. Wir nennen $f(a_1) \ldots f(a_n)$ das *Kodewort von $a_1 \ldots a_n$*.
Geben Sie für alle $m \geq 2$ eine Kodierung f_m von Σ_m durch Wörter über Σ_{bool} an derart, dass es eine Konstante c gibt so, dass für alle $m \geq 2$ und alle $a \in \Sigma_m$ gilt: $|f_m(a)| \leq c \cdot \log(m)$.

1.2 Algorithmische Probleme, Algorithmen

Wie zuvor erläutert, wird ein *algorithmisches Problem* in unseren Betrachtungen durch eine Wortfunktion beschrieben (gewünschtes Eingabe-Ausgabe-Verhalten eines eventuell unbekannten Algorithmus).

Beispiele algorithmischer Probleme:

1. Zu zwei natürlichen Zahlen in Dezimaldarstellung bilde das Produkt (wieder in Dezimaldarstellung).

   Beispiele: $8, 12 \mapsto 96$ $\qquad$ $001, 123 \mapsto 123$

   Zugehörige Wortfunktion: $f_1 : (\{0, \ldots, 9\}^*)^2 \to \{0, \ldots, 9\}^*$.
   (Wir müssen strenggenommen auch das leere Wort einbeziehen, setzen also z.B. $f_1(\epsilon, \cdot) = f_1(\cdot, \epsilon) = \epsilon$. Dies unterstellen wir analog auch in den weiteren Beispielen.)

2. Zu einer natürlichen Zahl finde ihre Primfaktorzerlegung.

   Beispiel: $120 \mapsto 2|2|2|3|5$

   $f_2 : \{0, \ldots, 9\}^* \mapsto \{0, \ldots, 9, |\}^*$.
   (Ein vom Berechnungsaufwand her sehr schwieriges, in der Kryptographie eingesetztes Problem.)

3. Zu einer positiven, rationalen Zahl finde die Wurzel, falls diese rational ist, sonst liefere keine Ausgabe.

   Beispiele: $36/100 \mapsto 3/5$ $\qquad$ $2 \mapsto \perp$

   $f_3 : \{0, \ldots, 9, /\}^* - \to \{0, \ldots, 9, /\}^*$.
   (Konvention: Bilde Wörter, die keine rationale Zahl darstellen, auf ϵ ab.)

4. Zu zwei natürlichen Zahlen teste, ob sie teilerfremd sind.

   Beispiele: $20, 21 \mapsto 1$(„ja“) $\qquad$ $20, 25 \mapsto 0$(„nein“)

   $f_4 : (\{0, \ldots, 9\}^*)^2 \to \{0, 1\}$.

5. Zu einem Polynom $p(x_1, \ldots, x_n)$ über $\mathbb{Z}$ teste, ob es eine Nullstelle $(z_1, \ldots, z_n) \in \mathbb{Z}^n$ gibt.

   Beispiele: $3x_1 + x_2 - {x_1}^2{x_2}^2 \mapsto 1$ (denn $(z_1, z_2) = (0, 0)$ ist Nullstelle).
   ${x_1}^2 + 1 \mapsto 0$.

Als Wörter schreiben wir Polynome etwa wie in LaTeX:

```
3x_1+x_2-{x_1}^2{x_2}^2
```

Damit wird das Problem beschrieben durch eine Funktion
$f_5 : \{0, \dots, 9, +, -, \_, \hat{}, \{, \}\}^* \to \{0, 1\}$.
(Dies ist das berühmte „10. Hilbertsche Problem“, das 1900 formuliert wurde und 1971 von Matiyasevich als algorithmisch unlösbar nachgewiesen wurde.)

6. Zu einem (Modula-)Programm mit Integer-Eingabevariable teste, ob es für Eingabe 0 terminiert.

   Zugehörige Wortfunktion: $f_6 : \Sigma^*_{\text{tastatur}} \to \{0, 1\}$.
   (Auch dieses Problem ist nicht algorithmisch lösbar, wie wir in Kapitel 3 zeigen werden.)

7. Zu einem endlichen, gerichteten Graph G und verschiedenen Knoten s, t von G finde einen kürzesten Weg von s nach t, falls ein solcher existiert, sonst gebe ϵ aus.

   Beispiel:

$$G: \quad \begin{tikzcd} 1 \arrow[rr] \arrow[dr] & & 2 \arrow[dr] & \\ & 3 \arrow[ur] \arrow[rr] & & 4 \end{tikzcd} \qquad s{=}1,\ t{=}4$$

   Ausgabe z.B. (1,3,4).

   Kodierung von G durch Wort über $\{0, \dots, 9, (,), -\}^*$.
   Anzahl der Knoten, Kantenliste, Knoten s, t: $4(1-2)(1-3)(3-2)(2-4)(3-4)(1)(4)$.
   Ausgabe $(1-3)(3-4)$. Unser Problem wird also beschrieben durch eine Wortfunktion

   $f_7 : \{0, \dots, 9, (,), -\}^* \to \{0, \dots, 9, (,), -\}^*$.
   (Ein effizient lösbares Problem; vgl. Kapitel 4)

Die Probleme 4,5,6 verlangen nur die Antwort „ja“/„nein“. Man spricht dann von *Entscheidungsproblemen.* Diese sind jeweils am bequemsten darstellbar durch eine *Relation.*

Beispiel: In Problem 4 betrachten wir
$R := \{(u, v) \in (\{0, \dots, 9\}^*)^2 | u, v \text{ Dezimaldarstellung zweier teilerfremder Zahlen}\}$

Allgemein gehen wir von einer Funktion $f : (\Sigma_1^*)^n \to \{0, 1\}$ über zur Relation

$$R = \{(w_1, \dots, w_n) \in (\Sigma_1^*)^n | f(w_1, \dots, w_n) = 1\}$$

Besteht dieser Zusammenhang zwischen f und R, nennt man f auch die *charakteristische Funktion* von R.

Nachdem wir *algorithmische Probleme* durch *Wortfunktionen* (und Entscheidungsprobleme auch durch Relationen) beschrieben haben, wenden wir uns den *Lösungen*, also den *Algorithmen* zu.

Ein **Algorithmus** ist ein Verfahren, das

- Eingabewörter schrittweise verarbeitet,
- durch endlichen Text bis ins letzte Detail eindeutig festgelegt ist,
- bei Termination eine Ausgabe liefert, oder nicht terminiert.

Definition 1.2.1 (berechenbare Funktion)
Ein Algorithmus $\mathfrak{A}$ berechnet die Funktion $f : (\Sigma_1^*)^n - \to \Sigma_2^*$, *falls er bei Vorlage von* $(w_1, \ldots, w_n) \in (\Sigma_1^*)^n$ *terminiert genau in dem Fall, dass* $(w_1, \ldots, w_n) \in Def(f)$, *und in diesem Fall als Ausgabe* $f(w_1, \ldots, w_n)$ *liefert.*
Eine Funktion f heißt (im intuitiven Sinne) berechenbar, *wenn solch ein Algorithmus existiert.*

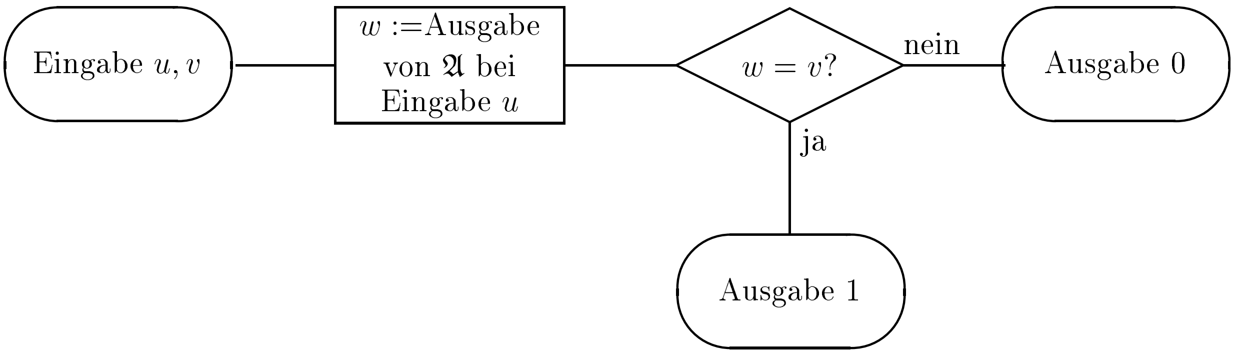
Definition 1.2.2 (entscheidbare Relation)
Ein Algorithmus $\mathfrak{A}$ entscheidet die Relation $R \subseteq (\Sigma_1^*)^n$, *falls er bei Vorlage von* $(w_1, \ldots, w_n) \in (\Sigma_1^*)^n$ *jeweils terminiert, und zwar mit Ausgabe 1 im Falle* $(w_1, \ldots, w_n) \in R$, *mit Ausgabe 0 im Falle* $(w_1, \ldots, w_n) \notin R$.
R heißt (im intuitiven Sinne) entscheidbar, *wenn solch ein Algorithmus existiert.*

Es gibt einen einfachen Zusammenhang zwischen „berechenbar“ und „entscheidbar“.
Hierzu betrachten wir den Fall $n = 1$ und gehen von einer Funktion $f : (\Sigma_1^*)^n - \to \Sigma_2^*$ zu ihrem „Graphen“ $R_f := \{(u, v) \in \Sigma_1^* \times \Sigma_2^* \mid f(u) = v\}$ über.

Satz 1.2.1 *Sei $f : \Sigma_1^* \to \Sigma_2^*$ total. Dann gilt:*

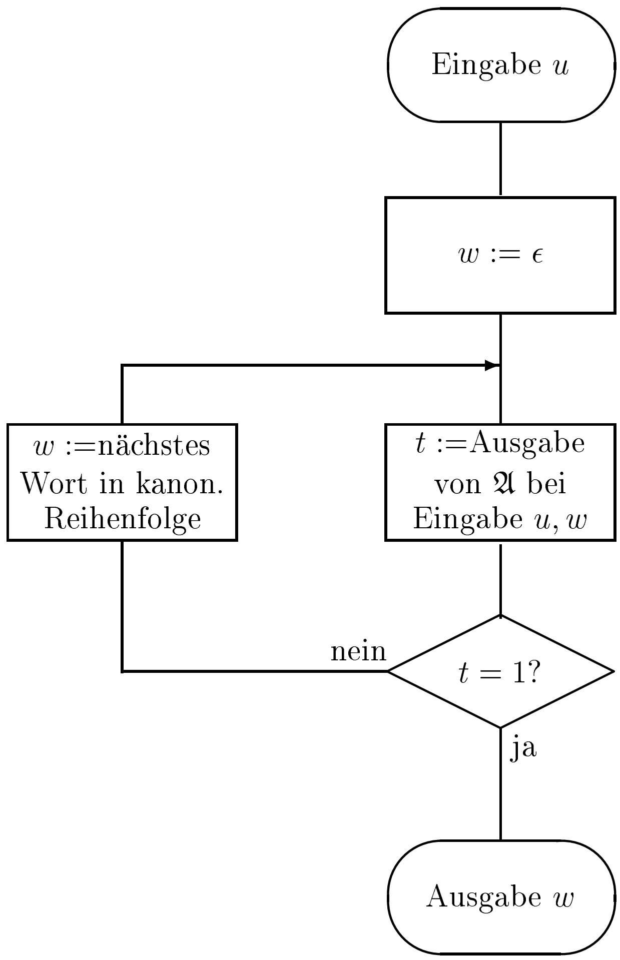
$$f \text{ berechenbar} \Leftrightarrow R_f \text{ entscheidbar.}$$

Beweis: „$\Rightarrow$“ Der Algorithmus $\mathfrak{A}$ berechne f. Folgender Algorithmus $\mathfrak{B}$ entscheidet dann R_f.



$\mathfrak{B}$ überprüft also nur, ob für ein vorgelegtes Paar (u, v) tatsächlich $f(u) = v$ gilt. Dies geht mit Rückgriff auf $\mathfrak{A}$, da $\mathfrak{A}$ ja f berechnet und f total ist.

„$\Leftarrow$“ $\mathfrak{A}$ entscheide R_f. Folgender Algorithmus $\mathfrak{B}$ berechnet f.



$\mathfrak{B}$ geht zu vorgelegtem u alle Paare (u, w) durch, bis eines gefunden ist so, dass $(u, w) \in R_f$. *Solch ein w gibt es, weil f total ist.* Die Überprüfung, ob $(u, w) \in R_f$, kann mit dem vorausgesetzten Algorithmus $\mathfrak{A}$ erfolgen.

Frage: Existiert ein analoger Satz für den Fall *partieller Funktionen* $f : \Sigma_1^* - \to \Sigma_2^*$ statt totaler Funktionen? Das führt uns auf den Begriff der *Semi-Entscheidbarkeit.*

Definition 1.2.3 (semi-entscheidbare Relation)
Ein Algorithmus $\mathfrak{A}$ semi-entscheidet *die Relation* $R \subseteq \Sigma_1^* \times \ldots \times \Sigma_n^*$, *falls er zur Eingabe* $(w_1, \ldots, w_n) \in \Sigma_1^* \times \ldots \times \Sigma_n^*$ *jeweils stoppt mit Ausgabe* 1 *(„er akzeptiert die Eingabe“), falls* $(w_1, \ldots, w_n) \in R$, *und* sonst nicht terminiert. *(Die Antwort „nein“ wird also nicht durch eine Ausgabe realisiert!)*

R heißt semi-entscheidbar, *falls ein Algorithmus existiert, der R semi-entscheidet.*

Beispiel: (Bsp. 6 s.o.) Die Menge der Modula-Programme mit Integer-Eingabevariablen, die für die Eingabe 0 stoppen, ist semi-entscheidbar:

Für ein Eingabeprogramm P verfolgt der Semi-Entscheidungsalgorithmus den Lauf von P auf Eingabe 0 und akzeptiert, falls P stoppt. Stoppt P nicht, bricht der Test wie gewünscht nicht ab.

Wir stellen für die drei grundlegenden Begriffe „f berechenbar“, „R entscheidbar“, „R semi-entscheidbar“ noch zwei Ergebnisse vor, die Zusammenhänge aufzeigen.

Zunächst zum Zusammenhang „entscheidbar“ – „semi-entscheidbar“ (formuliert für Wortmengen $W \subseteq \Sigma^*$ statt Wortrelationen $R \subseteq \Sigma_1^* \times \ldots \times \Sigma_n^*$):

Satz 1.2.2 *Eine Wortmenge $W \subseteq \Sigma^*$ ist entscheidbar gdw. W ist semi-entscheidbar und $\Sigma^* \backslash W$ ist semi-entscheidbar.*

Beweis: „$\Rightarrow$“ : Der Algorithmus $\mathfrak{A}$ entscheide W. $\mathfrak{A}'$ entstehe aus $\mathfrak{A}$ durch Übergang in Endlosschleife bei $\mathfrak{A}$-Stop mit Ausgabe 0. Dann ist klar: $\mathfrak{A}'$ semi-entscheidet W.
$\mathfrak{A}''$ entstehe aus $\mathfrak{A}$ durch Übergang in

- Endlosschleife bei $\mathfrak{A}$-Stop mit Ausgabe 1 („Eingabe $\in W$“),
- Stoppzustand mit Ausgabe 1 bei $\mathfrak{A}$-Stop mit Ausgabe 0 („Eingabe $\notin W$“).

Dann ist wiederum klar: $\mathfrak{A}''$ semi-entscheidet $\Sigma^* \backslash W$.

Zur Umkehrung „$\Leftarrow$“ : $\mathfrak{A}_1$ semi-entscheide W. $\mathfrak{A}_2$ semi-entscheide $\Sigma^* \backslash W$.
Folgender Algorithmus entscheidet dann W: Bei Eingabe $w \in \Sigma^*$ lasse $\mathfrak{A}_1, \mathfrak{A}_2$ auf Eingabe w abwechselnd jeweils einen Schritt weiter laufen und

(1) falls $\mathfrak{A}_1$ stoppt mit $\mathfrak{A}_1$-Ausgabe 1, dann stoppe und gebe 1 aus.

(2) falls $\mathfrak{A}_2$ stoppt mit $\mathfrak{A}_2$-Ausgabe 1, dann stoppe und gebe 0 aus.

Die Ausgabe ist korrekt (und der Algorithmus terminiert), da für $w \in W$ der Fall (1) und für $w \notin W$ der Fall (2) erfüllt wird.

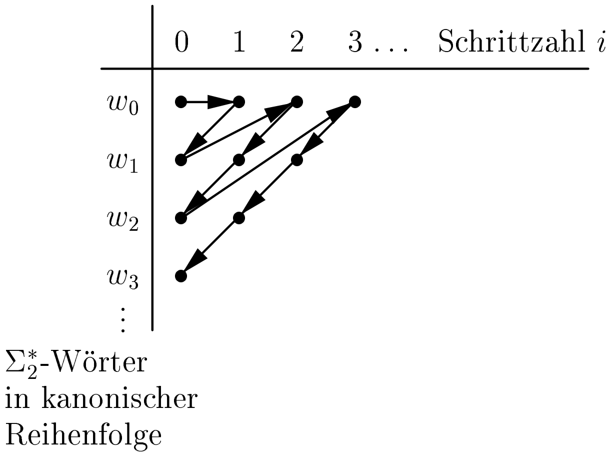
$\square$

Satz 1.2.3 *Eine Funktion $f : \Sigma_1^* - \to \Sigma_2^*$ ist berechenbar gdw. der zugehörige Graph R_f ist semi-entscheidbar.*

Beweis: „$\Rightarrow$“ : $\mathfrak{A}_f$ berechne f. Folgender Algorithmus $\mathfrak{B}$ semi-entscheidet dann R_f. Zu Eingabe $(u, v) \in \Sigma_1^* \times \Sigma_2^*$ wende $\mathfrak{A}_f$ auf u an und bei Stop von $\mathfrak{A}_f$ vergleiche $\mathfrak{A}_f$-Ausgabe $(= f(u))$ mit v. Falls dies zutrifft, stoppe mit Ausgabe 1, sonst terminiere nicht.

„$\Leftarrow$“ : $\mathfrak{A}_{R_f}$ semi-entscheide R_f. Folgender Algorithmus berechnet f: Lasse $\mathfrak{A}_{R_f}$ auf Eingabe (u, w) jeweils j Schritte lang laufen, für jedes Wort $w \in \Sigma_2^*$ und jede Schrittzahl j. Stoppt $\mathfrak{A}_{R_f}$ z.B. auf (u, v) nach j_0-Schritten mit Ausgabe 1 („$(u, v) \in R_f$“) so stoppe mit Ausgabe v und sonst laufe ohne Termination weiter.

Dieser Beweis nutzt also ein Schema zur Erfassung aller Paare (w, i):



$\square$

Eine alternative Fassung des Begriffs „semi-entscheidbar“ ist „aufzählbar“.

Definition 1.2.4 (Aufzählungsalgorithmus) *Ein* Aufzählungsalgorithmus $\mathfrak{A}$ *wird gestartet ohne Eingabe und liefert nach und nach Ausgabewörter (keines, endlich viele, unendlich viele) in irgendeiner Reihenfolge, evtl. mit Wiederholungen.*
Ist W die Menge der ausgegebenen Wörter, so sagen wir: $\mathfrak{A}$ zählt W auf.
Die Menge W heißt aufzählbar, falls ein Aufzählungsalgorithmus existiert, der W aufzählt.

Satz 1.2.4 *Es gilt (vgl. Übungen): W aufzählbar $\Leftrightarrow$ W semi-entscheidbar.*

Übungen

Übung 5 Zeigen Sie: Eine Wortmenge ist genau dann aufzählbar, wenn sie semi-entscheidbar ist.

Übung 6 (a) Historische Anmerkung: Die Goldbach'sche Vermutung (GBV) besagt, dass jede gerade Zahl ≥ 4 Summe zweier Primzahlen ist. Bisher konnte die GBV weder bewiesen noch widerlegt werden.

Zeigen Sie, dass die Funktion $f : \Sigma_{\text{bool}}^* \to \Sigma_{\text{bool}}^*$ mit

$$f(w) = \begin{cases} 1 & \text{falls GBV trifft zu} \\ 0 & \text{sonst} \end{cases}$$

berechenbar ist.

(b) Wählen Sie ein Spiel X unter den Spielen Dame, Mühle, Schach, Go. Eine X-*Konfiguration* ist eine Abbildung von der Menge der X-Felder in die Menge der X-Figuren. Ein X-*Partiepräfix* ist eine endliche Folge von X-Konfigurationen, die mit der Anfangskonfiguration beginnt und bei der sich jede Konfiguration aus dem vorangegangenen X-Partiepräfix gemäß den X-Regeln ergibt. Die X-Partiepräfixe denken wir uns durch geeignete Wörter über Σ_{bool} kodiert; sie bilden eine entscheidbare Wortmenge. Ein X-Partiepräfix heiße X-*Gewinnstellung für Weiß*, falls Weiß von dort aus einen Sieg erzwingen kann.

Zeigen Sie (für das X Ihrer Wahl), dass die Funktion $f : \Sigma_{\text{bool}}^* \to \Sigma_{\text{bool}}^*$ mit

$$f(w) = \begin{cases} 1 & \text{falls } w \text{ kodiert } X\text{-Gewinnstellung für Weiß,} \\ 0 & \text{sonst} \end{cases}$$

berechenbar ist.

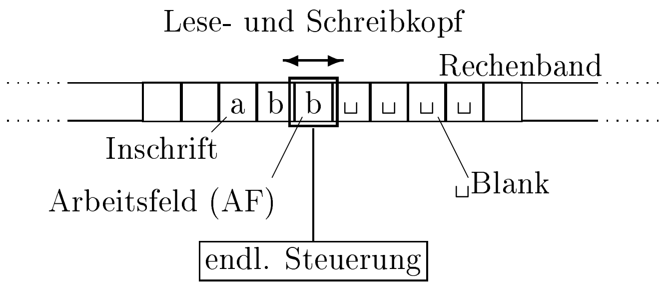
1.3 Definition der Turingmaschine

In einer für die Entwicklung der Informatik fundamentalen Arbeit hat Alan Turing 1936 (damals 24 Jahre alt) ein abstraktes Automatenmodell vorgeschlagen mit dem Ziel, die Durchführung beliebiger Algorithmen zur Symbolmanipulation in einem präzisen Rahmen zu erfassen. Turing analysierte hierzu die elementaren Einzelschritte, die ein Mensch bei der Realisierung eines Algorithmus (beim Rechnen auf dem Papier) durchführt. Er begründete einige wesentliche Grundannahmen: Der „Computer“ (in Turings Sinne der menschliche Rechner) hat nur eine feste endliche Anzahl interner Zustände, er arbeitet auf einem (ohne Beschränkung der Allgemeinheit) eindimensionalen Rechenpapier („Rechenband“), das in Felder geteilt ist und auf jedem Feld ein Symbol aus einem festen endlichen Zeichenvorrat aufnehmen kann. In jedem Augenblick übersieht der „Computer“ nur ein Segment gewisser fester Länge auf dem Rechenband, kann in Abhängigkeit von dessen Inschrift und seinem eigenen internen Zustand Änderungen in diesem Segment vornehmen und dann die Aufmerksamkeit auf benachbarte Felder des Rechenbandes verlegen. Auf kleinste Schritte reduziert, kann man sogar annehmen, dass das kritische Segment nur aus einem einzigen „Arbeitsfeld“ auf dem Rechenband besteht. In einem Schritt kann dann jeweils in Abhängigkeit vom internen Zustand und von der Inschrift des Arbeitsfeldes dort ein neues Symbol eingetragen werden, worauf das Arbeitsfeld dann eventuell um ein Feld nach rechts oder links verlegt wird.
Turings Analyse in seinen eigenen Worten:

> Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume that the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares.
> The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite.
> The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be "arbitrarily close" and will be confused. Again, the restriction is not one which seriously affects computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape.

> Let us imagine the operations performed by the computer to be split up into "simple operations" which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape. We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer (possibly with a special order), and the state of mind of the computer.
> We may suppose that in a simple operation not more than one symbol is altered. Any other changes can be split up into simple changes of this kind.

Da die Rechnungen nicht an mangelndem Platz scheitern sollen, wird das Rechenband als beidseitig unendlich vorausgesetzt. Allerdings sind in jedem Augenblick nur endlich viele Felder davon beschrieben, denn anfangs ist das Band leer, abgesehen von den endlich vielen Feldern, auf denen die Eingabe steht, und in einem Schritt kann jeweils höchstens ein weiteres Feld beschrieben werden. Formal repräsentiert man die „leere Beschriftung“ eines Feldes durch ein besonderes Leerzeichen, das „blank“, welches immer zum Zeichenvorrat dazugehört. Dies führt auf das Algorithmenmodell der „Turing-Maschine“, mit folgender Struktur:
Turing-Maschine:



Definition 1.3.1 (Turingmaschine)
Eine Turingmaschine *hat die Gestalt* $M = (Q, \Sigma, \Gamma, q_0, q_s, \delta)$ *mit:*

- *endlicher Zustandsmenge* Q*,*
- *Eingabealphabet* Σ *(wobei* $\llcorner\!\lrcorner \in \Sigma$*),*
- *Arbeitsalphabet* $\Gamma \supseteq \Sigma \cup \{\llcorner\!\lrcorner\}$*,*
- *Anfangszustand* $q_0 \in Q$*,*
- *Stoppzustand* $q_s \in Q$ *(wobei* $q_s \neq q_0$*),*
- *Übergangsfunktion* $\delta : (Q\backslash\{q_s\}) \times \Gamma \to Q \times \Gamma \times \{L, R, N\}$*.*

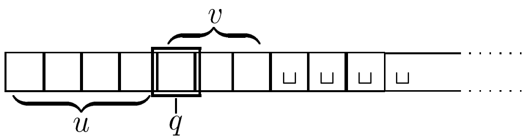
Bedeutung von $\delta(q, a) = (q', a', L/R/N)$: In Zustand q mit a auf Arbeitsfeld gehe in Zustand q' über, überschreibe a durch a' und bewege den Kopf ein Feld nach links/rechts/gar nicht. Notation durch Quintupel in der Turing-Befehlszeile: $q\ a\ q'\ a'\ L/R/N$

Zur Arbeitsweise: Zu jedem Zeitpunkt sind fast alle Felder des Rechenbandes mit $\llcorner\!\lrcorner$ beschrieben. Eine „Rechensituation“ wird erfasst durch den Begriff der „Konfiguration“.

Definition 1.3.2 *Eine* Γ-Konfiguration *ist gegeben durch:*

- *Inschrift u links von AF,*
- *Zustand q,*
- *Inschrift v von AF nach rechts bis zu einer Stelle, von der an nur $\sqcup$ auftritt.*

Schreibweise: uqv



Bemerkung 1.3.1 uqv *und* $uqv\sqcup$ *beschreiben die gleiche Konfiguration;*
uq *und* $uq\sqcup$, $uq\sqcup\sqcup$ *beschreiben die gleiche Konfiguration.*

$u'q'v'$ heißt *Folgekonfiguration* von uqv (kurz: $uqv \vdash_M u'q'v'$), falls einer der folgenden Fälle vorliegt (mit $v = av_0$ bzw. $v = \epsilon$, $a = \sqcup$):

$$\begin{array}{llllll}
(1) & \delta(q,a) = (q',a',L), & u = u_0b, & u' = u_0, & v' = ba'v_0, & u_0bq'a'v_0; \\
(2) & \delta(q,a) = (q',a',N), & u = \epsilon, & u' = \epsilon, & v' = a'v_0, & u_0q'ba'v_0; \\
(3) & \delta(q,a) = (q',a',R), & & u' = ua', & v' = v_0, & uq'a'v_0; \\
(4) & \delta(q,a) = (q',a',N), & & u' = a, & v' = a'v_0, & ua'q'v_0.
\end{array}$$

Von Konfiguration K wird Konfiguration K' erreicht (kurz: $K \vdash^*_M K'$), falls es eine Folge $K_0, K_1, \ldots, K_n$ (für ein $n \geq 0$) gibt mit $K_0 = K$, $K_n = K'$, $K_i \vdash_M K_{i+1}$ für $i < n$.
Eine *Stoppkonfiguration* hat die Form uq_sv. Ihr *Ergebnis* (Ausgabe) ist das längste Anfangsstück von v, das kein $\sqcup$ enthält.

Beispiel: $ab\sqcup q_s \underbrace{aab}\sqcup\sqcup a$ Ergebnis: aab

Definition 1.3.3 *Die Turingmaschine M* berechnet *die Funktion $f : \Sigma_1^* - \rightarrow \Sigma_2^*$, falls M (mit Eingabealphabet Σ_1) für jedes $w \in \Sigma_1^*$ von der Konfiguration q_0w aus eine Stoppkonfiguration erreicht genau dann, wenn $w \in Def(f)$, und in diesem Fall ist das Ergebnis der Stoppkonfiguration das Wort $f(w)$.*
M entscheidet *die Wortmenge $W \subseteq \Sigma_1^*$, falls M von q_0w aus jeweils eine Stoppkonfiguration erreicht mit Ergebnis 1 für $w \in W$, Ergebnis 0 für $w \in \Sigma_1^* \backslash W$ (d.h. $0, 1 \in \Gamma$).*
M semi-entscheidet *die Wortmenge $W \subseteq \Sigma_1^*$, falls M für $w \in \Sigma_1^* \backslash W$ von q_0w aus nicht stoppt und sonst mit Ausgabe 1 terminiert.*

Entsprechend definieren wir: *M berechnet $f : (\Sigma_1^*)^n \rightarrow \Sigma_2^*$*
M entscheidet $R \subseteq (\Sigma_1^*)^n$ durch Rückgriff auf Anfangskonfiguration $q_0w_1\sqcup w_2\sqcup \ldots \sqcup w_n$ für Eingabetupel $(w_1, \ldots, w_n)$.

q_0	$0/1$	q_1	$\underline{0}/\underline{1}$	R
q_0	$\sqcup$	q_s	0	N
q_1	$0/1$	q_1	$0/1$	R
q_1	$\sqcup$	q_2	0	N
q_2	$0/1$	q_2	$0/1$	L
q_2	$\underline{0}/\underline{1}$	q_s	$0/1$	N

Tabelle 1.1: Turingtafel für M (Definiton von δ)

Definition 1.3.4 (Turing-Berechenbarkeit, Turing-Entscheidbarkeit) *Eine Funktion f heißt* Turing-berechenbar *(Turing computable), wenn es eine Turingmaschine gibt, die f berechnet.*
Eine Relation R heißt Turing-entscheidbar *(recursive bzw. Turing-decidable), wenn es eine Turingmaschine gibt, die R entscheidet.*
Eine Relation R heißt Turing-semi-entscheidbar *(recursively enumerable), wenn es eine Turingmaschine gibt, die R semi-entscheidet.*

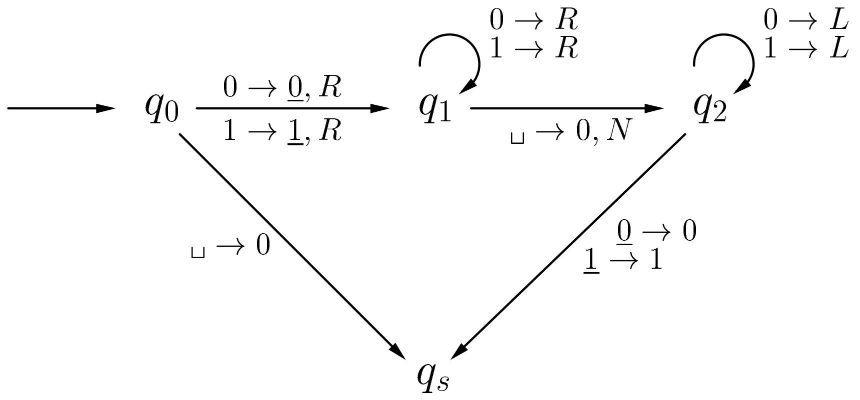
Beispiel 1: Die Funktion $f : \Sigma_{bool}^* \to \Sigma_{bool}^*$ mit $f(w) = w0$ ist Turing-berechenbar.
Wir wählen die folgende Turingmaschine: $M = (Q, \Sigma_{bool}, \Gamma, \delta, q_0, q_s)$ mit $Q = \{q_0, q_1, q_2, q_s\}$, $\Gamma = \{0, 1, \sqcup, \underline{0}, \underline{1}\}$, δ gemäß Tabelle 1.1.

Arbeitsweise von M anhand typischer Konfigurationen:
$q_0011 \underset{M}{\vdash} q_1\underline{0}11 \overset{*}{\underset{M}{\vdash}} \underline{0}11q_1\sqcup \underset{M}{\vdash} \underline{0}11q_20 \overset{*}{\underset{M}{\vdash}} q_2\underline{0}110 \underset{M}{\vdash} q_s0110$.

M wandelt das erste Eingabesymbol b in $\underline{b}$ um, geht in Zustand q_1 nach rechts auf das erste $\sqcup$, druckt dort 0 und geht (in q_2) zurück auf $\underline{b}$, wandelt es in b zurück und stoppt.

Konvention: Fehlt für ein Paar (q, a) die Zeile $qa\ldots$, so denken wir uns qaq_saN ergänzt.

Graphische Darstellung der Turingmaschine:



Konvention: Rechts von $\to$ werden nicht neugedruckte Buchstaben und Bewegungen N unterschlagen.

Beispiel 2: Wir betrachten das Alphabet $\Sigma = \{1\}$ und schreiben $\underline{n}$ für $|^n = \underbrace{|\ldots|}_{\text{n-mal}}$

Die *unäre Multiplikation* $f : \Sigma^* \times \Sigma^* \to \Sigma^*$ mit $f(\underline{n_1}, \underline{n_2}) = \underline{n_1 \cdot n_2}$ ist Turing-berechenbar.

Idee: $q_0 \,|||\sqcup||||\, \overset{*}{\underset{M}{\vdash}} \dots q_s\, ||||||||||||\sqcup \dots$
Kopiere den 2. Block so oft hinter den Input, wie es Striche im ersten Block gibt. Markiere abgearbeitete Striche in 1. Block durch $\nmid$.

Zwischenkonfiguration: $\nmid\nmid q \,| \sqcup |||| \sqcup ||||||||$

Details:

1. Falls $|$ auf AF, drucke $\nmid$, weiter bei 2.
   Sonst ($\sqcup$ auf AF), gehe zum ersten $\sqcup$ nach rechts, ein weiteres Feld nach rechts und stoppe.

2. Gehe auf Feld nach dem nächsten $\sqcup$ und $(*)$ kopiere den dort beginnenden Strichblock ans Ende des darauffolgenden Strichblocks.

3. Gehe zurück auf das letzte Feld mit $\nmid$, gehe ein Feld nach rechts und zurück nach 1.

Zu $(*)$: Ausgangskonfiguration: $\nmid\nmid \,| \sqcup p \,|||| \sqcup |||||||$

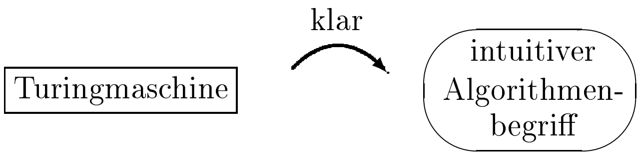
(a) Falls AF$=|$, überschreibe durch $\not|$ und setzte bei (b) fort. Sonst (AF$=\sqcup$) verwandle nach links gehend die $\not|$ zurück in $|$. Gehe nach 3.

(b) Gehe nach rechts auf zweites vorkommendes $\sqcup$ und überschreibe dies durch $|$.

(c) Gehe zurück auf Feld nach letztem $\not|$, weiter bei (a).

□

Wie weit werden durch Turingmaschinen die allgemeinen Algorithmen im intuitiven Sinne erfasst? Es ist klar, dass jede Turingmaschine auch einen Algorithmus im intuitiven Sinne darstellt:



Also: Jede Turing-berechenbare Wortfunktion ist auch im intuitiven Sinne berechenbar. (Denn eine Turingmaschine ist ein Algorithmus im intuitiven Sinne, von sehr spezieller Form.) Analoges gilt für Turing-Entscheidbarkeit und Turing-Semi-Entscheidbarkeit.

Die **Church-Turing-These** (1936) behauptet die Umkehrung: **Jede im intuitiven Sinne berechenbare Funktion ist Turing-berechenbar.**

Argumente dafür:

1. Turings Analyse des Rechenvorgangs.

2. Erfahrung seit über 60 Jahren. (konkrete Beispiele, Kombination gegebener Algorithmen zu neuen Algorithmen ist auch mit Turingmaschinen nachvollziehbar)

3. Erweiterungen des Turingmaschinen-Modells sind auf das Basismodell reduzierbar.

4. Andere, z.B. programmiersprachliche Fassungen des Algorithmenbegriffs, sind im präzisen Sinne äquivalent zu Turingmaschinen.

Anwendungen der Church-Turing-These (CTT):

- unwesentliche Anwendung:

  Ist ein Algorithmus in Skizzenform (Pseudocode, Umgangssprache) gegeben, so lässt er sich auch in die Form einer Turingmaschine bringen. (Diese Anwendung der CTT ist letztlich durch Fleißarbeit vermeidbar.)

- wesentliche Anwendung:

  Um zu zeigen, dass eine Funktion im intuitiven Sinne *nicht* berechenbar ist, genügt es nachzuweisen, dass sie nicht Turing-berechenbar ist. Hierfür lernen wir in Kapitel 3 Beispiele kennen.

Übungen

In den folgenden beiden Aufgaben sind Turingmaschinen anzugeben. Erläutern Sie vor der Definition der Turingmaschine zunächst ihre Arbeitsweise in der Umgangssprache, und geben Sie auch typische Konfigurationen aus einem Lauf der Turingmaschine an.

Übung 7 Geben Sie eine Turingmaschine an, die die Menge $\{w \in \Sigma_{\text{bool}}^* \mid w$ ist Binärdarstellung ohne führende Nullen einer positiven durch vier teilbaren natürlichen Zahl$\}$ entscheidet.

Übung 8 Geben Sie eine Turingmaschine an, die die „Spiegelbildfunktion" ^{R} über dem Alphabet Σ_{bool} berechnet.

Übung 9 Geben Sie eine Turingmaschine an, die die Funktion $f : \Sigma_{\text{bool}}^* - \to \Sigma_{\text{bool}}^*$ mit

$$f(w) = \begin{cases} 1 & \text{falls } |w| \text{ gerade,} \\ \perp & \text{sonst} \end{cases}$$

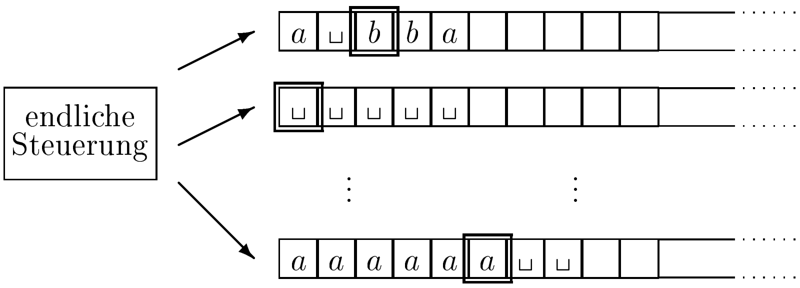
berechnet.

Übung 10 Sei M eine TM mit Eingabealphabet Σ_1, Startzustand q_0 und Stoppzustand q_s. Für eine Stoppkonfiguration $\alpha = uq_sv$ (mit $u, v \in \Gamma^*$ und v endet nicht auf $\sqcup$) von M sei die *Bandausgabe von* α definiert als uv.
M berechnet die Funktion $f : \Sigma_1^* - \to \Sigma_2^*$ *mit Bandausgabe*, falls erstens M für jedes $w \in \Sigma_1^*$ von Startkonfiguration q_0w aus genau dann stoppt, wenn $w \in \text{Def}(f)$, und zweitens in diesem Fall die Bandausgabe dieser Stoppkonfiguration das Wort $f(w)$ ist.
Zeigen Sie: Falls $f : \Sigma_1^* - \to \Sigma_2^*$ Turing-berechenbar ist, so gibt es eine Turingmaschine, die f mit Bandausgabe berechnet.

Hinweis: Geben Sie eine Konstruktion an, um aus einer TM $\mathfrak{A}$ eine TM $\mathfrak{B}$ zu gewinnen so, dass $\mathfrak{B}$ dieselbe Funktion mit Bandausgabe berechnet wie $\mathfrak{A}$ mit der in der Vorlesung gebrauchten Ausgabekonvention. Begründen Sie die Richtigkeit ihrer Konstruktion.

1.4 Mehrband-Turingmaschinen

Die Idee wird durch folgendes Diagramm erläutert:



k Bänder mit

k Schreib-/Leseköpfen

Befehlszeilen haben die Form $q(a_1, \dots, a_k)q'(a'_1, \dots, a'_k)\overbrace{(d_1, \dots, d_k)}^{\text{„Directions“}}$, wobei $\underbrace{d_i \in \{L, R, N\}}_{\text{Kopfbewegung}}$

Wirkung einer solchen Befehlszeile:
Vom Zustand q mit Buchstaben $a_1, \dots, a_k$ unter den k Köpfen gehe in Zustand q', überdrucke a_i durch a'_i und bewege den i-ten Kopf um d_i (nach links/rechts/nicht). $(*)$

Nun zu den präzisen Festlegungen:

Definition 1.4.1 (k-Band-Turingmaschine) *Eine k-Band-Turingmaschine hat die Form*

$$M = (Q, \Sigma, \Gamma, q_0, q_s, \delta)$$

mit $Q, \Sigma, \Gamma, q_0, q_s$ wie zuvor für 1-Band-Turingmaschinen und

$$\delta : (Q \backslash \{q_s\}) \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, N\}^k.$$

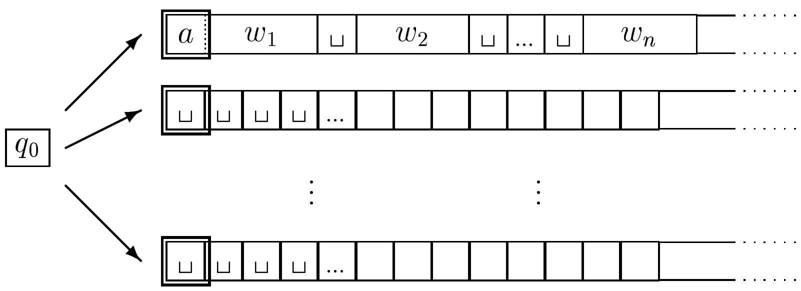
Eine Konfiguration *hat die Form $(q, u_1, a_1v_1, u_2, a_2v_2, \dots, u_k, a_kv_k)$, wobei $u_ia_iv_i$ die Inschrift des i-ten Bandes bis zu einer Stelle, von der an nur ␣ auftritt, und das Arbeitsfeld das erste Feld nach u_i ist (mit Beschriftung a_i).*

Wie früher definieren wir gemäß $(*)$:

$$K \underset{M}{\vdash} K'.$$

K' ist die Folgekonfiguration von K.

Die Anfangskonfiguration bei Eingabe $(w_1, \dots, w_n)$ sei festgelegt durch
$(q_0, \epsilon, w_1 {\sqcup} w_2 {\sqcup} \dots {\sqcup} w_n, \epsilon, {\sqcup}, \dots, \epsilon, {\sqcup})$



Stoppkonfigurationen sind wie früher definiert, mit Ausgabe auf 1. Band.
Mit diesen Festlegungen definieren wir für k-Band-Turingmaschinen M die Beziehung „M berechnet f“ analog zu 1-Band-Turingmaschinen.

Wir zeigen nun, dass wir mit k-Band-Turingmaschinen keine prinzipiell neuen Berechnungsmöglichkeiten erhalten.

Satz 1.4.1 (Reduktion von k-Band- auf 1-Band-Turingmaschinen) *Ist f durch eine k-Band-Turingmaschine berechenbar, so auch durch eine (1-Band)-Turingmaschine.*

Beweis: Sei k-Band-Turingmaschine $M = (Q, \Sigma, \Gamma, q_0, q_s, \delta)$ gegeben, die f berechnet. Wir geben eine (1-Band-)Turingmaschine M' an, die M „simuliert“:
Idee: Repräsentiere M-Konfiguration $(q, u_1, a_1v_1, u_2, a_2v_2, \ldots, u_k, a_kv_k)$ durch folgende M'-Konfiguration:

$$\hat{q}\#u_1\underline{a_1}v_1\#u_2\underline{a_2}v_2\#\ldots\#u_k\underline{a_k}v_k\#$$

mit neuen Symbolen $\underline{a}$ und $\#$ für $a \in \Gamma$.

Nun zur präzisen Formulierung:
Definition von M':

- Zustände: q_0', q_s', sowie $\hat{q}$ für jedes $q \in Q$, und weitere Hilfzustände
- Eingabealphabet Σ wie zuvor
- Arbeitsalphabet Γ' enthält die Buchstaben von Γ, außerdem $\underline{a}$ für $a \in \Gamma$, $\#$, und weitere Hilfzustände.

Arbeitsweise von M' bei Eingabe $w_1 \sqcup \ldots \sqcup w_n$ (Hinweise zur Definition der Übergangsfunktion δ'):

1. Herstellung der Konfiguration $\hat{q_0}\#\overbrace{\underline{a}w}^{w_1}\sqcup w_2\sqcup\ldots\sqcup w_n\#\underline{\sqcup}\#\ldots\#\underline{\sqcup}\#$ durch: Verschiebung der Inschrift $w_1\sqcup\ldots\sqcup w_n$ (= Inschrift bis zum n-ten $\sqcup$) um ein Feld nach rechts, Drucken anschließend von $\#\underline{\sqcup}\#\ldots\#\underline{\sqcup}\#$ (k-mal $\#$), Drucken von $\#$ vor $w_1\sqcup\ldots\sqcup w_n$, schließlich Übergang in $\hat{q_0}$).
   Sonderfall $w_1 = \epsilon$ ist extra zu behandeln.

2. Simulation eines M-Schrittes:
   Von $\hat{q}\#x_1\#\ldots\#x_k\#$ gehe nach rechts zum $(k+1)$-ten $\#$ und speichere die dabei gelesenen unterstrichenen Buchstaben $\underline{a_1},\ldots,\underline{a_k}$: Zustandsname $[q,a_1,\ldots,a_k]$.
   Rückkehr zum ersten $\#$ mit Modifikation der Bandinschrift gemäß der zugehörigen M-Befehlszeile $q(a_1,\ldots,a_k)q'(a'_1,\ldots a'_k)(d_1,\ldots,d_k)$:
   Ersetzen von $\underline{a_i}$ durch a'_i, Neusetzen oder Unterstreichen gemäß d_i (für $i=k,\ldots,1$) [Sonderfall $d_i = R$ und $\#$ rechts vom bisherigen Arbeitsfeld auf i-tem Band: Einfügen von $\sqcup$, unterstreichen, zuvor Rechtsverschiebung der Inschrift rechts vom Arbeitsfeld], gehe auf erstes $\#$ in M'-Zustand $\hat{q}'$.
   Für jede Kombination $q_0,a_1,\ldots,a_k$ gesonderter Block von M'-Befehlen mit gesonderten Zuständen.

3. Ausgabe: Übergang von $\hat{q}_s\#\ldots\underline{b}v\#\ldots\#$ zu $\#\ldots q'_s bv\#$
   Von $\hat{q}_s$ aus Übergang nach rechts zum zweiten $\#$, Ersetzen durch $\sqcup$, Rückkehr zum ersten unterstrichenen Buchstaben $\underline{b}$, Ersetzung durch ununterstrichenes b und Übergang in q'_s (Stoppzustand von M').

$\square$

Kapitel 2

WHILE- und GOTO-Programme

Turingmaschinen definieren ein Berechnungsmodell, das bitweise (oder buchstabenweise) arbeitet. Für allgemeine Analysen (etwa zur prinzipiellen Leistungsfähigkeit oder zu Leistungsgrenzen des Berechnungsmodells) ist dieser Zugang geeignet, nicht jedoch zur Formulierung konkreter Algorithmen. Hierzu wurden programmiersprachliche Formalismen (Programmiersprachen) eingeführt, die die Darstellung von Algorithmen verständlicher machen. Wir werden in diesem Kapitel zwei derartige Sprachen einführen und mit dem Modell der Turingmaschine vergleichen: die Sprache der `WHILE`-Programme (einem Kern von Pascal entsprechend) und die Sprache der `GOTO`-Programme (ein Mini-Fragment von Basic).
Als Datenbereich dient die Menge $\mathbb{N}$ der natürlichen Zahlen, und diese werden in elementaren Schritten auch als *Ganzes* verändert (also nicht in Form von Bitwörtern dargestellt und auf die Bitebene verändert).
Dadurch treten die Kontrollstrukturen der Programmiersprachen besser in den Vordergrund. Beim Vergleich mit den Turingmaschinen muss allerdings die Darstellung der natürlichen Zahlen (als Strichfolgen oder Bitfolgen) berücksichtigt werden.
Zur Vorbereitung führen wir zahlentheoretische Grundfunktionen ein und stellen einige Sachverhalte zur fundamentalen Methode der induktiven Definitionen und Beweise zusammen.

2.1 Funktionen über $\mathbb{N}$, induktive Definitionen

Wir benutzen folgende arithmetische Basisfunktionen (ohne auf Dezimal- oder Bit-Darstellung natürlicher Zahlen zurückzugreifen).

- Nachfolger $k \to k+1$
- Vorgänger $k \to k \dot{-} 1 = \begin{cases} k-1 & \text{, falls } k > 0 \\ 0 & \text{, falls } k = 0 \end{cases}$
- Summe $(k,l) \to k+l$, wie üblich
- Produkt $(k,l) \to k \cdot l$, wie üblich
- Differenz $(k,l) \to k \dot{-} l = \begin{cases} k-l & \text{, falls } k \geq l \\ 0 & \text{, falls } k \leq l \end{cases}$
- Quotient $(k,l) \to k$ div l hier definiert durch $\begin{cases} \lfloor \frac{k}{l} \rfloor & \text{, falls } l > 0 \\ 0 & \text{, falls } l = 0 \end{cases}$

- Rest von $(k,l) \to k$ und l hier definiert durch $\begin{cases} \text{Rest von } k \text{ div } l & , \text{ falls } l > 0 \\ 0 & \text{sonst} \end{cases}$

Wir wollen für die folgenden Überlegungen die Totalität der Basisfunktionen verlangen; daher legen wir auch einen Wert bei Division durch 0 fest.

Vereinbarung zur Einsetzung und Iteration von Funktionen:

Für $g_1, \dots, g_m : A- \to B, f : B^m- \to C$ sei die durch Einsetzung entstehende Funktion $h : A- \to C$ definiert durch $h(a) = f(g_1(a), \dots, g_m(a))$. Hierbei sei $f(g_1(a), \dots, g_m(a))$ undefiniert, falls ein $g_i(a)$ undefiniert ist oder alle $g_i(a)$ definiert sind, etwa mit den Werten $g_i(a) = b_i$, und $f(b_1, \dots, b_m)$ undefiniert ist.

Für $f : A- \to A$ sei die n-fache Iteration $f^n : A- \to A$ definiert durch $f^0 = id_A, f^{n+1}(a) = f(f^n(a))$.

Induktive Definitionen und Beweise

Die **induktive Definition einer Menge** M erfolgt durch Angabe von:

1. Grundelementen von M,

2. Übergangsmöglichkeiten von gegebenen zu neuen Elementen (Redeweise: „m entsteht unmittelbar aus $m_1, \dots, m_k$“),

3. Restriktion auf die gemäß 1. und endlichmalige Anwendung von 2. entstehenden Elemente.

Beispiel 1: M = Menge der natürlichen Zahlen

1. 0 ist eine natürliche Zahl (0 wird als Grundelement eingeführt).

2. Ist n eine natürliche Zahl, so auch $n+1$ („$n+1$ entsteht unmittelbar aus n“).

Beispiel 2: M = Menge der Wörter über Σ

1. ϵ ist ein Wort über Σ (ϵ wird als Grundelement eingeführt).

2. Ist w ein Wort über Σ und $a \in \Sigma$, so ist auch wa ein Wort über Σ („wa entsteht, für $a \in \Sigma$, unmittelbar aus w“).

Ist eine Menge M induktiv definiert, kann man eine Funktion $f : M \to A$ ebenfalls induktiv definieren, dem Aufbau von M folgend („induktiv über den Aufbau von M“).
Voraussetzung ist, dass beim Aufbau von M ein Element m, das nicht Grundelement ist, jeweils nur auf eindeutige Weise unmittelbar aus Elementen $m_1, \dots, m_k$ entsteht. (Ist diese Eindeutigkeit nicht gegeben, entsteht i.a. keine eindeutige Festlegung der Funktionswerte).

Die **induktive Definition** einer Funktion $F : M \to A$ erfolgt durch

1. Festlegung von $F(m)$ für die Grundelemente von M,

2. Festlegung von $F(m)$ für m, das unmittelbar aus $m_1, \ldots, m_k$ entsteht, mit Rückgriff auf $F(m_1), \ldots, F(m_k)$ (wie gesagt, unter der Voraussetzung, dass m so auf eindeutige Weise aus $m_1, \ldots, m_k$ entsteht).

   Beispiel 1: Die Längenfunktion $l : \Sigma^* \to \mathbb{N}$ wird induktiv definiert durch

   1. $l(\epsilon) = 0$,
   2. $l(wa) = l(w) + 1$, für $w \in \Sigma^*, a \in \Sigma$.

Induktive Beweise

Eine zweite Anwendung des induktiven Aufbaus von M ist der Nachweis, dass alle Elemente von M eine gewisse Eigenschaft E haben, durch Induktion über den Aufbau von M. Ein solcher *induktiver Beweis* einer Eigenschaft E für alle Elemente von M besteht in zwei Schritten:

1. Induktionsanfang: Zeige, dass jedes Grundelement von M die Eigenschaft E hat.
2. Induktionsschritt: m entstehe unmittelbar aus $m_1, \ldots, m_k$. Unter der Annahme, dass $m_1,\ldots,m_k$ die Eigenschaft E haben (Induktionsvoraussetzung), zeige:
   m hat E (Induktionsbehauptung).

Beispiele über die Menge $\mathbb{N}$ der natürlichen Zahlen sind wohlbekannt. In diesen Abschnitten werden weitere Beispiele gebracht, wobei die Menge M eine induktiv definierte Programmiersprache ist (die Behauptung also eine Aussage über beliebige Programme der Sprache ist).
Diese Beweise durch **Induktion über den Aufbau der Programme** bilden eine fundamentale Methode, um zu allgemeinen Aussagen über Programmiersprachen zu gelangen.

2.2 WHILE-Programme

Die Definition einer Programmiersprache besteht aus zwei Teilen:

- der *Syntax*, d.h. der Festlegung der Menge der zugelassenen Programme als Ketten von Zeichen
- der *Semantik*, d.h. der Festlegung der Bedeutung der Programme als Datentransformationen

(In unserem Fall wird die Bedeutung eines Programmes jeweils eine Transformation von natürlichen Zahlen sein, also eine zahlentheoretische Funktion.)

Syntax der `WHILE`-Programme:
Variablen haben die Form `Xi` mit positiver Dezimalzahl `i:X1,X2,X3,...` . Eine *Wertzuweisung* hat eine der folgenden Formen (mit Dezimalzahlen `i,j,l`):
`Xi:=0`, `Xi:=Xj`, `Xi:=Xj+1`, `Xi:=Xj-1`, `Xi:=Xj op Xl` mit op $\in \{+, -, \cdot, \texttt{div}, \texttt{mod}\}$

Definition 2.2.1 (WHILE-Programme) *Die Menge der* `WHILE`*-Programme ist induktiv definiert gemäß folgender Regeln:*

1. *Jede Wertzuweisung ist ein* `WHILE`*-Programm.*

2. - *Sind* P_1, P_2 `WHILE`*-Programme, so auch* `begin` P_1`;`P_2 `end`.
   - *Sind* P_1, P_2 `WHILE`*-Programme und ist* `Xi` *Variable, so ist*

     `if Xi>0 then begin` P_1 `end else begin` P_2 `end`

     ein `WHILE`*-Programm.*
   - *Ist* P_1 *ein* `WHILE`*-Programm und* `Xi` *eine Variable, so ist auch*

     `while Xi>0 do begin` P_1 `end`

     ein `WHILE`*-Programm.*
   - *Ist* P_1 *ein* `WHILE`*-Programm und* `Xi` *eine Variable, so ist auch*

     `loop Xi begin` P_1 `end`

     ein `WHILE`*-Programm.*

Geschieht der Aufbau ohne `WHILE`-Schleife, so sprechen wir von einem `LOOP`-Programm.
Beispiel eines `WHILE`-Programms:

```
begin
 begin
  X3:=X1;
  while X3>0 do begin
   begin
   X2:=X2+X1;
   X3:=X3-X1;
   end;
  end;
 end;
 X1:=X2;
end
```

Semantik der `WHILE`-Programme:
Jedes Programm P bestimmt die zugehörige *semantische Funktion* $[\![P]\!] : \mathbb{N}^{\mathbb{N}_+} - \to \mathbb{N}^{\mathbb{N}_+}$. Wir gehen aus von der Menge $\mathbb{N}^{\mathbb{N}_+}$ aller Folgen $(k_1, k_2, \ldots)$. Eine solche Folge ist eine Belegung von `X1`, `X2`,... durch natürliche Zahlen.
Ein Programm-„Zustand“ ist also ein Vektor $(k_1, k_2, k_3, \ldots)$.
Die i-te Projektion pr_i von $(k_1, k_2, k_3, \ldots)$ sei definiert durch $\mathrm{pr}_i(k_1, k_2, k_3, \ldots) = k_i$.

Definition 2.2.2 (Semantik der WHILE-Programme) *Die Definition von* $[\![P]\!]$ *erfolgt induktiv über den Aufbau von* P.

Wertzuweisung:

- *Für* $P =$ `Xi:=0` *sei* $[\![P]\!](k_1, k_2, \ldots) = (k_1, \ldots, k_{i-1}, 0, k_{i+1}, \ldots)$.
- *Für* $P =$ `Xi:=Xj+1` *sei* $[\![P]\!](k_1, k_2, \ldots) = (k_1, \ldots, k_{i-1}, k_j + 1, k_{i+1}, \ldots)$.
- *Für* $P =$ `Xi:=Xj-1` *sei* $[\![P]\!](k_1, k_2, \ldots) = (k_1, \ldots, k_{i-1}, k_j \dot{-} 1, k_{i+1}, \ldots)$.

- *Für* $P =$ `Xi:=Xj` *sei* $[\![P]\!](k_1, k_2, \ldots) = (k_1, \ldots, k_{i-1}, k_j, k_{i+1}, \ldots)$.
- *Für* $P =$ `Xi:=Xj` *op* `Xl` *sei* $[\![P]\!](k_1, k_2, \ldots) = (k_1, \ldots, k_{i-1}, k_j \text{ op } k_l, k_{i+1}, \ldots)$.
- *Für* $P =$`begin` $P_1; P_2$ `end` *sei* $[\![P]\!](k_1, k_2, \ldots) = [\![P_2]\!]([\![P_1]\!](k_1, k_2, \ldots))$.
- *Für* $P =$ `if Xi>0 then begin` P_1 `end else begin` P_2 `end` *sei*
  $$[\![P]\!](k_1, k_2, \ldots) = \begin{cases} [\![P_1]\!](k_1, k_2, \ldots) & , \textit{ falls } k_i > 0 \\ [\![P_2]\!](k_1, k_2, \ldots) & , \textit{ sonst} \end{cases}$$
- *Für* $P =$ `while Xi>0 do begin` P_1 `end` *sei*
  $$[\![P]\!](k_1, k_2, \ldots) = \begin{cases} [\![P_1]\!]^m(k_1, k_2, \ldots) & , \textit{ für das kleinste } m \geq 0 \textit{ mit} \\ & \quad \mathrm{pr}_i[\![P_1]\!]^m(k_1, k_2, \ldots) = 0, \\ & \quad \textit{falls ein solches existiert.} \\ \textit{undefiniert} & , \quad \textit{sonst} \end{cases}$$
- *Für* $P =$ `loop Xi begin` P_1 `end` *sei* $[\![P]\!](k_1, k_2, \ldots) = [\![P_1]\!]^{k_i}(k_1, k_2, \ldots)$.

Wir legen nun mit Rückgriff auf $[\![P]\!]$ noch eine Ein-Ausgabe-Konvention fest:

Definition 2.2.3 *Sei* P *ein* `WHILE`*-Programm. Die* n-stellige durch P berechnete Funktion $f_P^{(n)}$ *ist definiert durch* $f_P^{(n)}(k_1, \ldots, k_n) = \mathrm{pr}_1([\![P]\!](k_1, \ldots, k_n, 0, 0, \ldots))$.

Beispiele zu semantischen Funktionen $[\![\ ]\!]$

1. **Beispiel:**
   P_1: `loop X1 begin` $\underbrace{\texttt{X1:=X1-1}}_{P_0}$ `end;`

   Behauptung:
   $$[\![P_1]\!](k_1, k_2, \ldots) = (0, k_2, \ldots)$$

   Beweis:
   $$[\![P_1]\!](k_1, k_2, \ldots) = [\![P_0]\!]^{k_1}(k_1, k_2, \ldots) \stackrel{(*)}{=} (0, k_2, \ldots)$$

   Zu $(*)$**:** Zeige $[\![P_0]\!]^i(k_1, k_2, \ldots) = (k_1 \dot{-} i, k_2, \ldots)$ für $i \geq 0$.
   (Dann sind wir fertig mit $i = k_1$.)

   Induktionsanfang: $i = 0 : [\![P_0]\!]^0(k_1, k_2, \ldots) = (k_1, k_2, \ldots) = (k_1 \dot{-} 0, k_2, \ldots)$, wie zu zeigen war.

   Induktionsschritt:

   $$\begin{aligned} [\![P_0]\!]^{i+1}(k_1, k_2, \ldots) &= [\![P_0]\!]([\![P_0]\!]^i(k_1, k_2, \ldots)) \\ &\stackrel{\text{I.V.}}{=} [\![P_0]\!](k_1 \dot{-} i, k_2, \ldots) \\ &= ((k_1 \dot{-} i) \dot{-} 1, k_2, \ldots) \\ &= (k_1 \dot{-} (i+1), k_2, \ldots) \end{aligned}$$

$\square$

2. Beispiel: P habe folgende Gestalt (mit Teilprogrammen $P_1, \ldots, P_4$):

$$P \begin{cases} \texttt{begin} & \\ P_1 & \texttt{loop X1 begin X1:=X1}\dot{-}\texttt{1 end;} \\ P_2 & \texttt{loop X2 begin } \underbrace{\texttt{loop X3 begin } \overbrace{\texttt{X1:=X1+1}}^{P_4} \texttt{ end}}_{P_3} \texttt{ end} \\ \texttt{end} & \end{cases}$$

Behauptung:

$$[\![P]\!](k_1, k_2, \ldots) = (k_2 \cdot k_3, k_2, k_3, \ldots)$$

Beweis:

$$\begin{array}{rcl} [\![P]\!](k_1, k_2, \ldots) & = & [\![P_2]\!]([\![P_1]\!](k_1, k_2, \ldots)) \\ & \overset{\text{Bsp. 1}}{=} & [\![P_2]\!](0, k_2, k_3, \ldots) \\ & = & [\![P_3]\!]^{k_2}(0, k_2, k_3, \ldots) \\ & = & ([\![P_4]\!]^{k_3})^{k_2}(0, k_2, k_3, \ldots) \\ & = & [\![P_4]\!]^{k_3 \cdot k_2}(0, k_2, k_3, \ldots) \\ & = & (k_2 \cdot k_3, k_2, k_3, \ldots) \end{array}$$

□

Wir erhalten also:

$$\begin{array}{lcll} f_P^{(1)}(k) & = & 0 & \text{für alle } k \\ f_P^{(2)}(k,l) & = & 0 & \text{für alle } k,l \\ f_P^{(3)}(k,l,m) & = & l \cdot m & \text{für alle } k,l,m \end{array}$$

3. Beispiel:
Q: $\texttt{while X1>0 do begin } \underbrace{\texttt{loop X2 begin X1:=X1}\dot{-}\texttt{1 end}}_{Q_0} \texttt{ end}$

Wir wollen $[\![Q]\!]$ bestimmen:
Wie zuvor erhalten wir $[\![Q_0]\!](k_1, k_2, \ldots) = (k_1 \dot{-} k_2, k_2, k_3, \ldots)$
und somit $[\![Q_0]\!]^n(k_1, k_2, \ldots) = (k_1 \dot{-} (n \cdot k_2), k_2, k_3, \ldots)$.

Also:

$$[\![Q]\!](k_1, k_2, \ldots) = \begin{cases} (0, k_2, k_3, \ldots) & \text{falls } k_2 > 0 \\ (k_1, k_2, k_3, \ldots) & \text{falls } k_1 = 0, k_2 = 0 \\ \perp & \text{falls } k_1 > 0, k_2 = 0 \end{cases}$$

Damit ergibt sich:

$$[\![Q_0]\!]^{k_2}(k_1, k_2, \ldots) = \begin{cases} (k_1 \dot{-} (n \cdot k_2), k_2, \ldots) & \text{für das kleinste } n, \textit{mit} \\ & \mathrm{pr}_1(k_1 \dot{-} (n \cdot k_2), k_2) = 0, \\ & \text{falls ein solches } n \text{ existiert} \\ \perp & \text{sonst.} \end{cases}$$

Wir erhalten:

$$f_Q^{(1)}(k) = \begin{cases} 0 & \text{falls } k = 0 \\ \perp & \text{falls } k > 0 \end{cases}$$

$$f_Q^{(2)}(k,l) = \begin{cases} 0 & \text{falls } k = 0 \text{ oder } l > 0 \\ \perp & \text{sonst.} \end{cases}$$

Definition 2.2.4 (WHILE-Berechenbarkeit) *Das* `WHILE`*-Programm P berechnet die n-stellige Funktion $f : \mathbb{N}^n - \to \mathbb{N}$, falls $f_P^{(n)} = f$. Eine Funktion $f : \mathbb{N}^n - \to \mathbb{N}$ heißt* `WHILE`-berechenbar, *falls es ein* `WHILE`*-Programm gibt, das f berechnet.*

Wir überlegen uns, dass wir mit einer Teilsprache $\texttt{WHILE}_0$ dieselben Funktionen berechnen können wie mit `WHILE`-Programmen.

Definition 2.2.5 ($\textbf{WHILE}_0$-Programme) $\texttt{WHILE}_0$*-Programme seien aufgebaut wie* `WHILE`*-Programme, jedoch mit keinen Wertzuweisungen außer solchen der Form* `Xi:=Xi+1` *und* `Xi:=Xi-1`.

Der Begriff der $\texttt{WHILE}_0$-Berechenbarkeit wird nun analog zum Begriff der `WHILE`-Berechenbarkeit eingeführt.

Satz 2.2.1 (von WHILE zu $\textbf{WHILE}_0$) *Jede* `WHILE`*-berechenbare Funktion über $\mathbb{N}$ ist* $\texttt{WHILE}_0$*-berechenbar.*

Beweis: $f : \mathbb{N}^n - \to \mathbb{N}$ werde berechnet mit `WHILE`-Programm P. Die Zahl $maxind(P)$ sei der maximale in P auftretende Variablenindex.
In der Auswertung von $[\![P]\!](k_1, \ldots, k_m, 0, \ldots)$ treten nur Terme $[\![Q]\!](l_1, l_2, \ldots)$ auf, mit $l_{m+1} = l_{m+2} = \ldots = 0$, wobei $m = \max\{n, maxind(P)\}$.
Im folgenden stehen $Y, Y_1, \ldots$ für die Variablen $X_{m+1}, X_{m+2}, \ldots$.
Es genügt nun zu zeigen:
Für jede Wertzuweisung R (ohne `+1, -1`) existiert ein $\texttt{WHILE}_0$-Programm R' mit

$$[\![R]\!](k_1, \ldots, k_m, 0, \ldots) = [\![R']\!](k_1, \ldots, k_m, 0, \ldots).$$

$R :=$	`Xi:=0`	Nehme R': `loop Xi begin Xi:=Xi-1 end`

$R :=$	`Xi:=Xj`	Wir bestimmen R' so: `loop Xj begin Y:=Y+1 end;`
		`loop Xi begin Xi:=Xi-1 end;`
		`loop Y begin Xi:=Xi+1 end;`
		`loop Y begin Y:=Y-1 end`

$R :=$	`Xi:=Xj+Xk`	`Y:=Xj` (wie oben); `loop Xk begin Y:=Y+1 end;` `Xi:=Y; Y:=0` (wie oben)

Analog verfährt man für `-`, `*`, `div`, `mod`.

□

Übungen

Übung 11 Ein `WHILE`$^-$-*Programm* ist ein `WHILE`-Programm, in dem kein `if .. then .. else`-Konstrukt vorkommt. Eine Funktion $f : \mathbb{N}^n - \to \mathbb{N}$ heißt `WHILE`$^-$-*berechenbar*, falls es ein `WHILE`$^-$-Programm gibt, das f berechnet.
Zeigen Sie: Jede `WHILE`-berechenbare Funktion ist `WHILE`$^-$-berechenbar.

Zusatzfrage: Gibt es zu jedem `WHILE`-Programm P ein `WHILE`$^-$-Programm P' mit $[\![P]\!] = [\![P']\!]$?

Übung 12 Sei $n \geq 1$ und $i, j, k \leq n$. Geben Sie $\texttt{WHILE}_0$-Programme P_a, P_b an derart, dass für alle $k_1, k_2, \ldots, k_n \in \mathbb{N}$ gilt

(a) $[\![P_a]\!](k_1, \ldots, k_n, 0, \ldots) = [\![\texttt{Xi:=Xj div Xk}]\!](k_1, \ldots, k_n, 0, \ldots)$,

(b) $[\![P_b]\!](k_1, \ldots, k_n, 0, \ldots) = [\![\texttt{Xi:=Xj mod Xk}]\!](k_1, \ldots, k_n, 0, \ldots)$.

Übung 13 Betrachte folgendes `WHILE`-Programm:

```
P :   X3:=X2;
      loop X3
      begin
        X1:=X1*X3;
        X3:=X3 ∸ 1
      end
```

Zeigen Sie, dass

$$[\![P]\!](k_1, k_2, \ldots) = (k_1 \cdot (k_2!), k_2, 0, k_4, \ldots)$$

für alle $k_1, k_2, \ldots \in \mathbb{N}$.

2.3 Vergleich von LOOP und WHILE

Die `loop`-Anweisung erlaubt eine a priori festgelegte Anzahl von Iterationen (siehe 2. Beispiel im vorangehenden Abschnitt zur Berechnung von „·“). Die `while`-Anweisung hingegen erlaubt eine unbeschränkte Wiederholung der Ausführung eines Statements bis zur Erfüllung eines Abbruchkriteriums.

Beispiel ($3x + 1$-Funktion): Sei $g(n) = \begin{cases} \frac{n}{2} & \text{falls n gerade} \\ 3n + 1 & \text{falls n ungerade} \end{cases}$

Die $3x + 1$-Funktion F ist definiert durch:

$$F(n) = \begin{cases} \text{kleinstes } k \text{ mit } g^k(n) \leq 1 & \text{falls ein solches } k \text{ existiert} \\ \perp & \text{sonst} \end{cases}$$

Zum Beispiel ergibt sich $F(11) = 14$, Die 15 g-Iterationswerte $g^0(11), \ldots, g^{14}(11)$ sind: 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.
In der Berechnung von F benutzt man entsprechend eine `WHILE`-Schleife. Ob man F auch ohne `WHILE` berechnen kann, ist eine bis heute offene Frage. Umgekehrt sieht man leicht:

Satz 2.3.1 *Jede* `LOOP`*-berechenbare Funktion ist* `WHILE`*-berechenbar ohne* `loop`*-Anweisung.*

Wir verwenden folgenden Hilfssatz:
Hilfssatz: Sei P `LOOP`-Programm, $m \geq maxind(P)$.
Dann existiert ein `WHILE`-Programm P' ohne `loop`-Anweisung, mit

$$(*) \qquad [\![P]\!](k_1, \ldots, k_m, 0, \ldots) = [\![P']\!](k_1, \ldots, k_m, 0, \ldots) \qquad .$$

Der Beweis des Satzes mit dem Hilfssatz ist nun leicht:
Sei $f : \mathbb{N}^n - \to \mathbb{N}$ `LOOP`-berechenbar, etwa durch P. Wende Hilfssatz auf P und $m = \max\{n, maxind(P)\}$ an. Wir erhalten P' ohne loop-Anweisung mit $(*)$. Da $n \leq m$, gilt $f_P^{(n)} = f_{P'}^{(n)}$.

Beweis des Hilfssatzes durch Induktion über den Aufbau der `LOOP`-Programme:

Falls P Wertzuweisung: Nehme $P' = P$.
Falls $P = P_1; P_2$; sei $m \geq maxind(P)$. Dann ist $m \geq maxind(P_1)$ und $m \geq maxind(P_2)$.
Die Induktionsvoraussetzung liefert P_1', P_2'. Setze $P' := P_1'; P_2'$. Überprüfung von $(*)$:

$$\begin{array}{rcl} [\![P]\!](k_1, \ldots, k_m, 0, \ldots) & = & [\![P_2]\!]([\![P_1]\!](k_1, \ldots, k_m, 0, \ldots)) \\ & \stackrel{\text{I.V.}}{=} & [\![P_2']\!]([\![P_1']\!](k_1, \ldots, k_m, 0, \ldots)) \\ & & [\![P']\!](k_1, \ldots, k_m, 0, \ldots) \end{array}$$

Falls $P =$ `if Xi` > 0 `then begin` P_1 `end else begin` P_2 `end` analog zu $P_1; P_2$.
Falls $P =$ `loop Xi begin` P_1 `end`, und $m \geq maxind(P)$ gegeben. Dann ist $m \geq maxind(P_1)$.
Wähle $\underline{P}_1$ gemäß I.V. Setze $j := \max\{m, maxind(\underline{P}_1)\} + 1$ und

$$\underline{P} : \texttt{Xj} := \texttt{Xi};\ \texttt{while Xj} > 0\ \texttt{do begin}\ \underline{P_1};\ \texttt{Xj} := \texttt{Xj} - 1\ \texttt{end}$$

Dann hat $\underline{P}$ kein `loop`, und $[\![\underline{P}]\!](k_1, \ldots, k_m, 0, \ldots) = [\![P]\!](k_1, \ldots, k_m, 0, \ldots)$.

□

Die Umkehrung dieses Satzes gilt nicht:

Satz 2.3.2 *Nicht jede* `WHILE`*-berechenbare Funktion ist auch* `LOOP`*-berechenbar.*

Hierzu genügt das
Lemma: Jede `LOOP`-berechenbare Funktion ist total.
Denn andererseits ist z.B. $f_Q^{(2)}$ (Beispiel 3) `WHILE`-berechenbar, aber nicht total.

Beweis des Lemmas: durch Nachweis, dass $[\![P]\!]$ total ist, durch Induktion über den Aufbau der `LOOP`-Programme P.
(Beachte: Die durch P berechnete n-stellige Funktion ist $f_P^{(n)}(\overline{k}) = \text{pr}_1[\![P]\!](k_1, \ldots, k_n, 0 \ldots)$, also mit $[\![P]\!]$ ebenfalls total.)
Falls P Wertzuweisung, so $[\![P]\!]$ total.
Falls $P = P_1; P_2$,
dann ist $[\![P]\!](k_1, \ldots) = [\![P_2]\!] \underbrace{[\![P_1]\!](k_1, \ldots)}_{\text{definiert nach I.V.}}$, etwa mit $[\![P_1]\!](k_1, \ldots) = (k_1', k_2', \ldots)$.
$[\![P_2]\!](k_1', k_2', \ldots)$ ist dann wiederum definiert nach I.V.

Der Fall `if...then...else` wird analog behandelt.
Für $P =$`LOOP Xi begin` P_1 `end` gilt $[\![P]\!](k_1, \ldots) = [\![P_1]\!]^{k_i}(k_1, \ldots)$. Also ist mit $[\![P_1]\!]$ auch $[\![P]\!]$ total. $[\![P_1]\!]$ aber ist total nach Induktionsvoraussetzung.

□

Gibt es *totale* Funktionen, die `WHILE`-berechenbar, aber nicht `LOOP`-berechenbar sind?
Die Antwort lautet „Ja“: Ein Beispiel ist die *Ackermann*-Funktion $A : \ \mathbb{N} \times \mathbb{N}$, definiert wie folgt:

(1) $A(0, y) = y + 1$

(2) $A(x + 1, 0) = A(x, 1)$

(3) $A(x+1,y+1) = A(x, A(x+1,y))$

Für jedes Zahlenpaar (x,y) ist genau eine Gleichung anwendbar: (1) bei $x = 0$, (2) bei $x > 0$ und $y = 0$, (3) bei $x > 0$, $y > 0$. Wir verfolgen eine Auswertung beginnend mit $(x,y) = (1,3)$:

$$\begin{aligned}
A(1,3) &= A(0,A(1,2)) = A(0,A(0,A(1,1))) \\
&= A(0,(A(0,A(0,A(1,0))))) \\
&= A(0,(A(0,A(0,A(0,1))))) \qquad \text{Regel (2)} \\
&= A(0,A(0,A(0,2))) \\
&= A(0,A(0,3)) \\
&= A(0,4) \\
&= 5
\end{aligned}$$

Bemerkung: *Die Anwendung von (1),(2),(3) führt für jedes Paar $(x,y) \in \mathbb{N} \times \mathbb{N}$ nach endlich vielen Schritten auf eine natürliche Zahl (kurz: „$A(x,y)$ definiert“).*
Beweis: Durch Induktion über x.
Behauptung für x: Für alle y ist $A(x,y)$ definiert.

Induktions-Anfang bei $x = 0$: $A(0,y) = y+1$, also definiert für alle y.
Induktions-Voraussetzung: $A(x,y)$ definiert für alle y.
Induktions-Behauptung: $A(x+1,y)$ definiert für alle y.

Hierzu führen wir für unser festes $x+1$ eine „induktion“ über y:

induktions-anfang bei $y = 0$: $A(x+1,0) \overset{(2)}{=} A(x,1)$ def. nach Ind.-Vor. (für $y = 1$).
induktions-voraussetzung: $A(x+1,y)$ ist definiert.
induktions-schritt: $\underbrace{A(x+1,y+1) \overset{(3)}{=} A(x, \underbrace{A(x+1,y)}_{\text{def. nach ind.-vor.}})}_{\text{def. nach Ind.-Vor.}}$

Insgesamt gilt: Für jedes Argument (x,y) ist $A(x,y)$ definiert.

□

Wir haben also gezeigt, dass die durch (1),(2),(3) definierte Funktion total und (nach dem Rechenschema der Einsetzungen) auch berechenbar ist.

Man kann überprüfen:
$$\begin{aligned}
&A(0,y) > y \\
&A(1,y) > y+1 \\
&A(2,y) > 2y \\
&A(3,y) > 2^y \\
&A(4,y) > \left.2^{2^{\cdot^{\cdot^{2}}}}\right\} y\text{-mal} \\
&A(5,y) > 10^{10000}
\end{aligned}$$

Satz 2.3.3 (von Ackermann) *A ist* `WHILE`*-berechenbar, aber nicht* `LOOP`*-berechenbar.*

Beweis: Die `WHILE`-Berechenbarkeit übergehen wir hier zunächst.
Zum 2. Teil stellen wir die Beweismethode vor. Man zeigt folgenden

Hilfssatz: Für jedes `LOOP`-Programm P und jedes $m \geq maxind(P)$ existiert eine Zahl $k_{P,m}$ mit

$$\max\{[\![P]\!](k_1, \ldots, k_m, 0, \ldots)\} < A(k_{P,m}, \max\{k_1, \ldots, k_m\}) \quad (*)$$

für alle $k_1, \ldots, k_m$.

Beweis: Durch mühsame Induktion über den Aufbau der `LOOP`-Programme.

Hiermit folgt die 2. Teilbehauptung des Satzes:
Annahme: Das `LOOP`-Programm P berechne A und es sei $m \geq$ maxind(P).
Wähle gemäß der formulierten Behauptung $k_{P,m}$. Dann gilt:

$$A(x,y) \overset{\text{P ber. A}}{=} f_P^{(2)}(x,y) \leq \max[\![P]\!](x,y,0,\ldots)) \overset{(*)}{<} A(k_{P,m}, \max(x,y,0,\ldots)) \text{ für alle } x,y.$$

Setzen wir $x = y = k_{P,m}$, so folgt ein Widerspruch!

□

Übungen

Übung 14 Ein `LOOP`-*freies Programm* ist ein `WHILE`-Programm, in dem kein `loop` und kein `while` vorkommt. Eine Funktion $f : \mathbb{N}- \to \mathbb{N}$ heißt `LOOP`-*frei berechenbar*, falls es ein `LOOP`-freies Programm gibt, das f berechnet.
Zeigen Sie: Es gibt eine totale, `LOOP`-berechenbare Funktion, die nicht `LOOP`-frei berechenbar ist.

Hinweis: Finden Sie eine geeignete Eigenschaft (der semantischen Funktion) von `LOOP`-freien Programmen so, dass Sie erstens per Induktion zeigen können, dass jedes `LOOP`-freie Programm P diese Eigenschaft hat, und dass Sie zweitens eine totale, `LOOP`-berechenbare Funktion angeben können, die nicht von einem Programm mit dieser Eigenschaft berechnet werden kann.

Übung 15 Zeigen Sie, dass folgende Funktionen loop-berechenbar sind:

(a) $f_1 : \mathbb{N} \to \mathbb{N}$, $f_1(n) = \sum_{i=1}^{n} i$,

(b) $f_2 : \mathbb{N} \to \mathbb{N}$, $f_2(n) = \lfloor \sqrt{n} \rfloor$.

Übung 16 Wir betrachten nochmals die Fragestellung der Übung 14, aber diesmal für *simple* Programme, das sind loopfreie Programme, in denen alle Zuweisungen von der Form `Xi:=Xi+1`, `Xi:=Xi-1` sind (, also keine Zuweisungen der Form `Xi:=Xj` op `Xk` für op $\in$ {`+`, `-`, `*`, `div`, `mod`} oder der Form `Xi:=Xj`, `Xi:=Xj+1`, `Xi:=Xj-1` (für `i`$\neq$`j`), oder der Form `Xi:=0` auftauchen.)
Zeigen Sie: Es gibt kein simples Programm, welches die Funktion $f : \mathbb{N} \to \mathbb{N}$, $f(n) = 0$ berechnet.

Wiederum der Hinweis: Finden Sie eine geeignete Eigenschaft der semantischen Funktion von simplen Programmen so, dass Sie erstens per Induktion zeigen können, dass die semantische Funktion jedes simplen Programmes P diese Eigenschaft hat, und dass zweitens für jede Funktion $F : \mathbb{N}^{\mathbb{N}_+} \to \mathbb{N}^{\mathbb{N}_+}$ mit dieser Eigenschaft $\mathrm{pr}_1 \circ F$ verschieden von f ist.

Übung 17 (a) Sei $\mathit{fib} : \mathbb{N} \to \mathbb{N}$ definiert wie folgt: $\mathit{fib}(0) = \mathit{fib}(1) = 1$ und $\mathit{fib}(n+2) = \mathit{fib}(n) + \mathit{fib}(n+1)$ für alle $n \in \mathbb{N}$.

Zeigen Sie: fib ist `LOOP`-berechenbar.

(b) Sei $f : \mathbb{N} \to \mathbb{N}$ `LOOP`-berechenbar, und sei $F : \mathbb{N}^2 \to \mathbb{N}$, $F(x, y) = f^y(x)$. (d.h. die y-fache Iteration von f, angewandt auf x.)

Zeigen Sie: F ist `LOOP`-berechenbar.

2.4 GOTO-Programme

Um die Brücke zwischen den (strukturierten) `WHILE`-Programmen und den (unstrukturierten) Turingmaschinen zu schlagen, führen wir als Zwischensprache die `GOTO`-Programme ein; diese haben bedingte Sprünge als wichtigste Kontrollstruktur.
Wir legen wiederum zunächst die Syntax und dann die Semantik fest.

Definition 2.4.1 (Syntax der `GOTO`-Programme) *Ein `GOTO`-Programm hat die Form*

$$P = 1\alpha_1;\ 2\alpha_2, \dots, k\alpha_k$$

mit $\alpha_k =$`stop` *und für* $1 \leq l < k$ *sei* $\alpha_l = \texttt{Xi} := \texttt{Xi} + \texttt{1}$ *oder* $\alpha_l = \texttt{Xi} := \texttt{Xi} - \texttt{1}$ *oder* $\alpha_l =$`if Xi=0 goto l'` *mit* $1 \leq l' \leq k$.
Zeilennummern werden in Dezimaldarstellung geschrieben.

Definition 2.4.2 (Semantik der `GOTO`-Programme) *Eine P-Konfiguration hat die Form* $(l, k_1, k_2, \dots,)$ *mit* $1 \leq l \leq k$, $k_1, k_2 \dots \in \mathbb{N}$ *(Nummer der auszuführenden Anweisung, Belegung der Variablen* `X1`, `X2`, ...*)*
Die Folgekonfiguration zu $(l, k_1, k_2, \dots)$ *ist:*

im Falle $\alpha_l =$ `Xi:=Xi+1` $\quad (l+1, k_1, \dots, k_{i-1}, k_i + 1, k_{i+1}, \dots)$

$\alpha_l =$ `Xi:=Xi-1` $\quad (l+1, k_1, \dots, k_{i-1}, k_i \dot{-} 1, k_{i+1}, \dots)$

$\alpha_l =$ `if Xi=0 goto l'` $\quad \begin{cases} (l', k_1, k_2, \dots) & \textit{falls } k_i = 0 \\ (l+1, k_1, k_2, \dots) & \textit{falls } k_i > 0 \end{cases}$

P *berechnet* die n-stellige Funktion $f : \mathbb{N}^n - \to \mathbb{N}$, falls gilt:
Von der Konfiguration $(1, k_1, \dots, k_n, 0, \dots)$ aus erreicht P schließlich eine Stoppkonfiguration $(k, k'_1, k'_2, \dots)$ genau dann, wenn $f(k_1, \dots, k_n)$ definiert ist, und dann ist $f(k_1, \dots, k_n) = k'_1$.
Die n-stellige Funktion, die P berechnet, werde mit $f_P^{(n)}$ bezeichnet.

Beispielprogramme:

P_1:
```
1 if X1=0 goto 4;
2 X1:=X1-1;
3 if X2=0 goto 1;
4 stop;
```

P_2:
```
1 if X1=0 goto 1;
2 stop;
```

P_3:
```
1 stop;
```

Zu P_1 verfolgen wir die ersten Konfigurationen, beginnend mit $(1, 2, 1, 0, 0 \ldots)$, d.h. mit Befehlszeile 1 und Variablenwerten $2, 1, 0, 0, \ldots$

$$\begin{array}{r|llll} 1 & 2 & 1 & 0 & 0 \\ 2 & 2 & 1 & 0 & 0 \\ 3 & 1 & 1 & 0 & 0 \\ 4 & 1 & 1 & 0 & 0 \end{array}$$

$f_{P_1}^{(2)}(2, 1) = 1$

Allgemein gilt: $f_{P_1}^{(2)}(k_1, k_2) = \begin{cases} 0 & k_1 = 0 \\ 0 & k_1 > 0, \quad k_2 \geq 0 \\ k_1–1 & k_1 > 0, \quad k_2 > 0. \end{cases}$

Zu P_2:
Es gilt $f_{P_2}^{(1)}(k_1) = \begin{cases} \perp & k_1 = 0 \\ k_1 & k_1 > 0 \end{cases}$

Zu P_3:
Es gilt $f_{P_3}^{(1)}(k_1) = k_1$, also $f_{P_3}^{(1)} = id_{\mathbb{N}}$.

Übungen

Übung 18 Bestimmen Sie die Funktionen $f_P^{(1)}$, $f_P^{(2)}$, $f_P^{(3)}$ und $f_P^{(4)}$ für das folgende Programm P:

```
 1 X1:=X1+1
 2 if X2 = 0 goto 6
 3 X2:=X2-1
 4 X1:=X1+1
 5 if X4=0 goto 2
 6 if X3=0 goto 10
 7 if X1=0 goto 10
 8 X1:=X1-1
 9 if X4 = 0 goto 7
10 stop
```

Übung 19 `GOTO`'-*Programme* seien wie `GOTO`-Programme definiert, jedoch ohne Sprunganweisungen der Form $\texttt{if}\,\texttt{Xj} = \texttt{0}\,\texttt{goto}\,\texttt{l}$, dafür aber mit Sprunganweisungen der Form $\texttt{if}\,\texttt{Xj} > \texttt{0}\,\texttt{goto}\,\texttt{l}$. Die Semantik ist in der naheliegenden Weise definiert. Eine Funktion $f : \mathbb{N}^n - \rightarrow \mathbb{N}$ ist `GOTO`'-*berechenbar* gdw. es ein `GOTO`'-Programm gibt, welches f berechnet. Sei $n \geq 1$ und $f : \mathbb{N}^n - \rightarrow \mathbb{N}$. Zeigen Sie: f ist `GOTO`-berechenbar gdw. f ist `GOTO`'-berechenbar.
Zusatzfragen (ohne Wertung): Gibt es zu jedem `GOTO`-Programm P ein `GOTO`'-Programm P' mit $[\![P]\!] = [\![P']\!]$? Wie verhält es sich mit der umgekehrten Frage?

2.5 Der Äquivalenzsatz

Ziel der kommenden Betrachtungen ist es zu zeigen, dass Turingmaschinen, `WHILE`- und `GOTO`-Programme dieselben Funktionen berechnen. Hierzu zeigen wir den folgenden Ringschluss für die eingeführten Berechenbarkeitsbegriffe:

$$\texttt{WHILE} \overset{\checkmark}{\to} \texttt{WHILE}_0 \to \texttt{GOTO} \to k\text{-Band TM} \overset{\checkmark}{\to} \text{TM} \to \texttt{WHILE}$$

Die mit ✓ gekennzeichneten Übergänge sind schon gezeigt (Satz 2.2.1, Satz 1.4.1).
Wir gehen schrittweise vor und zeigen zuerst:

Satz 2.5.1 *Jede* $\texttt{WHILE}_0$*-berechenbare Funktion ist* `GOTO`*-berechenbar.*

Beweis: Wir verfahren induktiv über den Aufbau der $\texttt{WHILE}_0$-Programme:
Zu jedem $\texttt{WHILE}_0$-Programm P und $m \geq \text{maxind}(P)$ konstruieren wir ein `GOTO`-Programm P' so, dass für $(k_1, \ldots, k_m)$:
$[\![P]\!](k_1, \ldots, k_m, 0, \ldots)$ ist definiert und $[\![P]\!](k_1, \ldots, k_m, 0, \ldots) = (j_1, \ldots, j_m, 0 \ldots)$ gdw.
P' stoppt angesetzt auf $k_1, \ldots k_m$ und liefert die Werte $(j_1, \ldots, j_m, 0 \ldots)$ als Ergebnis.

Induktionsanfang:
Für P : `Xi:=Xi ± 1` setzen wir:
P' : `1 Xi:=Xi ± 1`
`2 stop`

Induktionsschritt:
Fall 1: P habe die Form: $P_1; P_2$, und sei m gegeben mit $m \geq \text{maxind}(P)$.
Induktionsvoraussetzung ist anwendbar auf P_1, P_2. Erhalte P_1', P_2' mit l_1 bzw. l_2 Zeilen. P' sei zusammengesetzt aus P_1' ohne `stop` und P_2' mit um $l_1 - 1$ erhöhten Zeilennummern.
Fall 2: P : `if Xi>0 then` P_1 `else` P_2, m wie zuvor gegeben.
Die Induktionsvoraussetzung liefert `GOTO`-Programm P_1', P_2', mit l_1 bzw. l_2 Zeilen.
Fall 3: P : `while Xi>0 do begin` P_1 `end`.

P' :
```
1      if Xi=0 goto l1+2
```
P_1 mit um 1 erhöhten Zeilennummern, ohne `stop`
```
l1+1   if X(m+1)=0 goto 1
l1+2   stop
```

□

Von den `GOTO`-Programmen läßt sich leicht die Brücke zu Turingmaschinen schlagen, wenn wir für jede benutzte `GOTO`-Programmvariable ein eigenes Rechenband als entsprechenden Speicher benutzen:

Satz 2.5.2 *Jede* `GOTO`*-berechenbare Funktion ist für geeignetes* k *durch eine* k*-Band-TM berechenbar.*

Beweis: $f : \mathbb{N}^n - \to \mathbb{N}$ werde berechnet durch das `GOTO`-Programm mit Variablen `X1`, ... ,`Xm` mit l Zeilen, wir setzen $k := \max\{n, m\}$. Wir geben (informell) eine k-Band-TM $\mathfrak{A}$ an, die f berechnet. Wir beschreiben die Arbeitsweise für Eingabe $(k_1, \ldots, k_n)$. (Auf 1. Band steht dann $|^{k_1}\sqcup|^{k_2}\sqcup\ldots\sqcup|^{k_n}$.)

1) Vom Anfangszustand gehe über in Konfiguration mit den Bandinschriften

   $\$|^{k_1}, \quad \$|^{k_2}, \quad \ldots, \quad \$|^{k_n}, \quad \$, \quad \ldots, \quad \$$

   Dies geschieht durch Rechtsverschiebung auf 1. Band, Eintrag von $\$, n-1$ Kopiervorgänge der Blöcke $|^{k_i}$ auf jeweils Band Nummer i $(i = 2, \ldots, n)$. Arbeitsfeld jeweils auf dem letzten $|$, bzw. im Fall $k_i = 0$ auf \$. Übergang in TM-Zustand $\hat{q}_1$ (Zustand $\hat{q}_i$ ist zuständig für `GOTO`-Programmzeile Nummer i).

2) `GOTO`-Programmschrittsimulation

   Für `GOTO`-Programmzeile $j, j+1 \ldots$ führe einen Block von TM-Befehlen ein, beginnend mit Zustand $\hat{q}_j$.
   Fall j `Xi:=Xi+1`: Gehe auf i-tem Band 1 Feld nach rechts, drucke $|$ und gehe über in Zustand $\hat{q}_{j+1}$.

   Fall j `Xi:=Xi-1`: Lösche $|$ auf Arbeitsfeld des i-ten Bandes und gehe ein Feld nach links, bzw. falls kein $|$ vorliegt, tue nichts. Anschließend Übergang nach $\hat{q}_{j+1}$.

   Fall j `if Xi=0 goto j'`: Führe folgende TM-Befehle durch $\hat{q}_j \ \$ \ \hat{q}_{j'} \ \$ \ N$; $\hat{q}_j \mid \hat{q}_{j+1} \mid N$.

3) Aus Zustand $\hat{q}_l$ (entspricht Stoppzeile von P) gehe auf 1. Band auf das Feld nach \$ (richtige Ausgabe durch TM) und stoppe durch Übergang in Stoppzustand.

□

Nach Satz 1.4.1 ist eine k-Band-Turingmaschine durch eine 1-Band-Turingmaschine simulierbar. Um die Äquivalenz zwischen Turingmaschinen und `WHILE`-Programmen nachzuweisen, fehlt uns also nur noch ein einziger Schritt: Jede Turingmaschinen-berechenbare Funktion $f : \mathbb{N}^n - \rightarrow \mathbb{N}$ ist auch `WHILE`-berechenbar.

Dieser Schritt hat mit einem tiefliegenden Problem zu tun: Turingmaschinen-Programme sind (ebenso wie `GOTO`-Programme) i.a. „unstrukturiert“; ihre Sprungbefehle sind nicht irgendwelchen Einschränkungen unterworfen. Andererseits haben `WHILE`-Programme eine durch ihren induktiven Aufbau induzierte wohlstrukturierte Form; insbesondere sind Schleifen im Kontrollfluss entweder disjunkt oder ineinander enthalten (man kann nicht mitten aus einer `WHILE`-Schleife „herausspringen“). Eine Simulation von Turingmaschinen durch `WHILE`-Programme muss also eine Konstruktion enthalten, die unstrukturierte Programme zu wohlstrukturierten macht. Dies geschieht, indem wir ein Turingmaschinen-Programm durch ein `WHILE`-Programm „interpretieren“. Das erfordert u.a. eine Kodierung der TM-Konfigurationen durch Zahlen.

Satz 2.5.3 *Jede Turing-berechenbare Funktion über* $\mathbb{N}$ *ist auch* `WHILE`*-berechenbar.*

Beweis: Wir gehen aus von einer 1-Band-TM $\mathfrak{A}$ mit Zuständen $q_1, \ldots, q_m, q_0$, wobei zwecks besserer Übersicht q_1 der Anfangszustand und q_0 der Stoppzustand ist. Die Arbeitsalphabet-Buchstaben seien $a_0, a_1, \ldots, a_{n-1}$.
Idee: Kodiere eine TM-Konfiguration $\kappa : a_{i_k} \ldots a_{i_0} q_r a_{j_0} \ldots a_{j_k}$ durch ein Zahlenquadrupel $(Z_\kappa, L_\kappa, R_\kappa, AF_\kappa)$, dessen Komponenten den Zustand, die Bandinschrift links von AF, die Bandinschrift rechts von AF sowie die Arbeitsfeld-Nummer kodieren:

$$Z_\kappa = r$$

$$L_\kappa = i_0 + i_1 n + i_2 n^2 + \ldots + i_k n^k$$

$$R_\kappa = j_0 + j_1 n + j_2 n^2 + \ldots + j_k n^k$$

$$AF_\kappa = k + 1$$

Für L_κ, R_κ benutzen wir also die n-adische Zahldarstellung (jeder Alphabetbuchstabe induziert eine der Ziffern $0, \ldots, n-1$), wobei die kleinste Stelle beim AF liegt.

Beispiel: $n = 2$, Arbeitsalphabet $\{\sqcup, |\} = \{a_0, a_1\}$, $\kappa = |||\sqcup|q_{17}\sqcup|||$.
Dann ist $Z_\kappa = 17$, $L_\kappa = (11101)_2 = 29$, $R_\kappa = (1110)_2 = 14$, $AF_\kappa = 5$.

Im gesuchten `WHILE`-Programm (`WHILE`-Interpreter für TM'en) benutzen wir die Variablen Z, L, R, AF für die Werte Z_κ, L_κ, R_κ, AF_κ (offiziell könnten wir `X1,X2,X3,X4` nehmen).

Grobstruktur des `WHILE`-Programms $P_\mathfrak{A}$:

1) Überführung der Variablenwerte $k_1, \ldots, k_n$ (Argumente für Funktionsberechnung) in die Kodierung der TM-Ausgangskonfiguration:

$$q_1|^{k_1}\sqcup|^{k_2}\sqcup \cdots \sqcup|^{k_n}.$$

2) TM-Schrittsimulation: Solange $Z > 0$ (TM macht weiter), simuliere $\mathfrak{A}$-Zeile $q_r a_j \ldots$, wenn $Z = r$ und (`X3` mod n) $= j$.

3) Extraktion der Ausgabe (wenn $Z = 0$, erreicht die TM ihre Stoppkonfiguration) aus der vorliegenden TM-Konfiguration.

Beispiel:$\ldots q_0|^k\sqcup$ soll auf Wert k in `X1` führen.

Präzisierung:
1. Phase: für den Fall $n = 2$ (zweistellige Funktion).
Die `WHILE`-Programm-Hilfsvariable N enthält Wert n (Anzahl n der Buchstaben im Arbeitsalphabet herstellbar mit Anweisungen $\underbrace{N := 0, N := N+1, \ldots N := N+1}_{n\text{-mal}}$). Analog erlauben wir auch im folgenden Wertzuweisungen der Form `X`$:= i$.

$$AF := 0$$

$R :=$ Code von Inschrift $|^{X1}\sqcup|^{X2}$ (Eingabewerte `X1`, `X2`)

$$R := 1 + 1 \cdot N + 1 \cdot N^2 + \ldots + 1 \cdot N^{X1-1} + 0 + N^{X1} + 1 \cdot N^{X1+1} + \ldots + 1 + N^{X1+X2}$$

$$L := 0$$

$$Z := 1$$

R wird wie folgt berechnet:
$R := \underbrace{1 + 1 \cdot N \ldots + 1 \cdot N^{X_1 \dot{-} 1}}_{\texttt{loop}-\text{Schleife}} \underbrace{+1 \cdot N^{X_1+1} + 1 \ldots + 1 \cdot N^{X_1+X_2}}_{\texttt{loop}-\text{Schleife}}$

2. Phase: Schrittsimulation
`while Z>0 do`
`if Z=i and (R mod N)=j then`
  stelle gemäß Turingzeile $q_i, a_j \ldots$ neue Werte für L, R, Z, AF her. $(*)$

Ausführung: `if`...`then`... steht für `if`...`then`...`else X1:=X1`.

`if Z=i and (R mod N)=j then`... steht für:
```
   Y1:=I; Y2:=J;
   Y2=1-|Z-Y1|+|(R mod N)-Y2|
   if Y>0 then...
```

Beachte: Es gilt $|A - B| = (A \dot{-} B) + (B \dot{-} A)$

zu $(*)$**:** Wie können drei Fälle unterscheiden:

1. Fall: $q_i a_j q_i' a_j' R$ in Turingtafel für q_i, a_j:
   Beispiel: Arabisch-alphabetische Dezimalziffern $108q_{17}2|225 \rightsquigarrow 1081q_{18}25$ vermöge der Turing-Zeile $q_{17}2q_{18}1R$

   allgemein: $Z := i'$
   $L := L \cdot N + j'$
   $R := R \;\texttt{div}\; N$
   $AF := AF + 1$

2. Fall: $q_i a_j q_i' a_j' L$ in Turingtafel für q_i, a_j:

   a) `if AF=0 then begin Z:=i'; R=(R-j)+j' end`
   b) `Z:=i'; R:=((R-j)+j')+N+(L mod N)`
   `L:=L div N; AF:=AF-1 end`

3. Fall: $q_i a_j q_i' a_j' N$ in Turingtafel für q_i, a_j: analog zu a).

3. Phase anhand eines Beispiels: Für die Bandinschrift $\overbrace{1 \ldots 1}^{k}\sqcup$ vom Arbeitsfeld an muss in der Variablen X_1 der Wert k hergestellt werden. Hierzu muss in X_1 so oft 1 addiert werden, wie der Rest 1 bei der wiederholten Division von R durch n auftritt.
Dies führt auf folgendes Verfahren:

```
X1:=0
loop R begin
  if R mod N=1 then X1:=X1+1 else R:=0;
  R:=R div N
end
```

□

Wir fassen zusammen:

Satz 2.5.4 (Äquivalenzsatz) *Für eine Funktion f über den natürlichen Zahlen sind folgende Aussagen äquivalent:*

- *f ist Turing-berechenbar.*
- *f ist* `GOTO`*-berechenbar.*
- *f ist* `WHILE`*-berechenbar.*

Zugleich sehen wir, dass sich die Church-Turing-These von Turingmaschinen auf `GOTO`- und `WHILE`-Programme überträgt. Das hat eine interessante Konsequenz:

Über den natürlichen Zahlen reichen die wenigen Konstrukte der `GOTO`*- bzw.* `WHILE`*-Programme letztlich aus, um beliebige Algorithmen darzustellen.*

Wir wollen diese Einsicht noch verschärfen, indem wir für `WHILE`-Programme analysieren, wie viele Schleifen (`loop` bzw. `while`) zur Berechnung beliebiger Funktionen investiert werden müssen. Hier ergibt sich das überraschende Ergebnis, dass man (etwa zur Berechnung zweistelliger Funktionen) nur eine *feste* Anzahl von Schleifen benötigt, darunter *nur eine einzige* `WHILE`*-Schleife*. Das sieht man so ein: Von einem beliebigen `WHILE`-Programm ausgehend, stellt man gemäß dem Ringschluß dieses Abschnitts nacheinander ein äquivalentes `GOTO`-Programm, eine äquivalente Turingmaschine und daraus wieder ein äquivalentes (nun stark normiertes) `WHILE`-Programm her. Dieses hat folgende Schleifenstruktur:

1. Initialisierung: Feste Anzahl der `LOOP`-Schleifen gemäß Stellenzahl.
2. Schrittsimulation: Eine `WHILE`-Schleife und eine feste Anzahl von `LOOP`-Schleifen.
3. Ergebnisextraktion: Eine `LOOP`-Schleife.

Folgerung:

Satz 2.5.5 (Normalformensatz) *Jede* `WHILE`*-berechenbare Funktion $f : \mathbb{N}^n - \rightarrow \mathbb{N}$ ist durch ein* `WHILE`*-Programm mit nur einer einzigen* `WHILE`*-Schleife und mit einer Anzahl von* `LOOP`*-Schleifen berechenbar, die nur von der Stelligkeit n abhängt.*

Es gibt einen weiteren, mathematisch sehr eleganten Formalismus zur Darstellung genau der (Turing- oder `WHILE`-)berechenbaren Funktionen über $\mathbb{N}$: Den *Kalkül der rekursiven Funktionen.*

Daher heißen die im präzisen Sinne berechenbaren Funktionen in der Literatur zumeist **rekursive Funktionen** (statt: Turing-, `GOTO`- oder `WHILE`-berechenbare Funktionen).
Wir skizzieren kurz die Definitionen: Eine Funktion über $\mathbb{N}$ heißt *rekursiv*, wenn man sie aus einfachen „Grundfunktionen" in endlich vielen Schritten mit Hilfe dreier Prozesse (Einsetzung, primitive Rekursion, Anwendung des μ-Operators) gewinnen kann. Diese Grundfunktionen und Prozesse sind wie folgt definiert:

Als Grundfunktionen sind zugelassen:

- die einstellige Funktion succ: $\mathbb{N} \to \mathbb{N}$ mit $\mathrm{succ}(x) = x + 1$,
- für $n \geq 1$ und $1 \leq i \leq n$ die i-te n-stellige Projektion $\mathrm{pr}_i^{(n)} : \mathbb{N}^n \to \mathbb{N}$ mit

$$\mathrm{pr}_i^{(n)}(x_1, \ldots, x_n) = x_i,$$

- für $n \geq 0$ und $k \geq 0$ die n-stellige Konstante $\mathrm{const}_k^{(n)}$ mit Wert k:

$$\mathrm{const}_k^{(n)}(x_1, \ldots, x_n) = k.$$

Die drei genannten Prozesse sind:

1. *Einsetzung:* Übergang von $g_1, \ldots, g_m : \mathbb{N}^n - \to \mathbb{N}$ und $h : \mathbb{N}^m - \to \mathbb{N}$ zur Funktion $f : \mathbb{N}^n - \to \mathbb{N}$ mit $f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$.

2. *Primitive Rekursion:* Übergang von $g : \mathbb{N}^n - \to \mathbb{N},\ h : \mathbb{N}^{n+2} - \to \mathbb{N}$ zu $f : \mathbb{N}^{n+1} - \to \mathbb{N}$, gegeben durch die Rekursionsgleichung $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$,
$f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, f(x_1, \ldots, x_n, y), y)$.
Diese Version von Rekursion lässt sich zur Definition vieler zahlentheoretischer Funktionen verwenden.

3. *Anwendung des μ-Operators:* Übergang von $g : \mathbb{N}^{n+1} - \to \mathbb{N}$ zu $f : \mathbb{N}^n - \to \mathbb{N}$, gegeben durch $f(x_1, \ldots, x_n) = \begin{cases} \text{kleinstes } y \text{ mit } g(x_1, \ldots, x_n, y) = 0 & \text{falls solches existiert,} \\ \perp & \text{sonst.} \end{cases}$

In den Fällen nicht totaler Funktionen sind genauere Überlegungen nötig, auf die wir hier aber nicht näher eingehen können. vgl. die Lehrbücher von Hermes, McNaughton und Wagner.

Übungen

Übung 20 Zeigen Sie (für den Fall $n = 1$): Wenn $g : \mathbb{N} \to \mathbb{N}$ und $h : \mathbb{N}^3 \to \mathbb{N}$ `LOOP`-berechenbar sind und f aus g, h durch primitive Rekursion entsteht, so ist f ebenfalls `LOOP`-berechenbar.

Hinweis: Aus zwei gegebenen `LOOP`-Programmen P_g, P_h, welche g bzw. h berechnen, ist also ein entsprechendees `LOOP`-Programm zu definieren, welches f berechnet.

Übung 21 Eine Funktion ist *primitiv rekursiv*, wenn sie eine der obigen Grundfunktionen ist oder aus primitiv rekursiven Funktionen vermöge der Prozesse Einsetzung und primitive Rekursion, d.h. ohne Anwendung des μ-Operators, entsteht.

Als Beispiel betrachte $f : \mathbb{N}^2 \to \mathbb{N}$, $f(x,y) = x + \sum_{i=0}^{y} i$. Man gewinnt f aus den Gleichungen

$$\begin{aligned} f(x,0) &= x & (=: g(x)) \\ f(x,y+1) &= f(x,y) + y + 1 & (=: h(x,y,f(x,y))). \end{aligned}$$

Wählt man also $g = \mathrm{id}_{\mathbb{N}}$ und $h : \mathbb{N}^3 \to \mathbb{N}$ with $h(x,y,z) = z + y + 1$, so entsteht f aus g und h durch primitive Rekursion.

Überlegen Sie sich, dass die Funktionen $\mathbb{N}^2 \to \mathbb{N}$, $(x,y) \mapsto x + y$ und $\mathbb{N}^2 \to \mathbb{N}$, $(x,y) \mapsto x \cdot y$ primitiv rekursiv sind.

(a) Zeigen Sie, dass $\mathrm{sig} : \mathbb{N} \to \mathbb{N}$ mit

$$\mathrm{sig}(x) = \begin{cases} 1 & \text{falls } x > 0, \\ 0 & \text{sonst,} \end{cases}$$

und die Vorgängerfunktion $V : \mathbb{N} \to \mathbb{N}$ mit $V(x) = x \dot{-} 1$ primitiv rekursiv sind.

(b) Seien $f_1, f_2 : \mathbb{N} \to \mathbb{N}$ primitiv rekursiv. Zeigen Sie, dass $f : \mathbb{N} \to \mathbb{N}$ definiert durch

$$f(x) = \begin{cases} f_1(x) & \text{falls } x > 0, \\ f_2(x) & \text{sonst,} \end{cases}$$

primitiv rekursiv ist.

Kapitel 3

Unentscheidbarkeit

Aufbauend auf der Präzisierung des Algorithmenbegriffs durch Turingmaschinen weisen wir nach, dass gewisse Probleme nicht algorithmisch lösbar sind. Es handelt sich einerseits um Probleme, die Algorithmen (Turingmaschinen) betreffen, andererseits um elementare kombinatorische Probleme.

3.1 Wortproblem für Turingmaschinen, Diagonalisierung

Terminologie: $M : w \to$ stop besage: TM M, angesetzt auf w, stoppt schließlich. Wir sagen auch: „M akzeptiert w." Erfolgt die Termination mit Ausgabe v: schreiben wir $M : w \to v$. Sonst $M : w \to \infty$.

Wortproblem für TM:
Gegeben: TM M (hier über dem Alphabet $\{0,1\}$), $w \in \{0,1\}^*$.
Frage: Gilt $M : w \to$ stop?

Satz 3.1.1 *Das Wortproblem für TM ist unentscheidbar. (Es gibt keinen Algorithmus, der zu beliebig vorgegebener TM entscheidet, ob die TM das Wort akzeptiert.)*

Wir zeigen den Satz mit Rückgriff auf die Church-Turing-These. Es genügt demnach, zu zeigen: Es gibt keine TM, die das Wortproblem entscheidet.

Eine TM, die das Wortproblem lösen soll, nimmt als Eingabe Paare der Form (TM, Wort) entgegen. Wir müssen daher TM'en als Eingaben für TM'en vorbereiten, d.h als Wörter kodieren.

Kodierung der Turingmaschinen (hier über $\Sigma = \{0,1\}$)

Zustände seien o.B.d.A. natürliche Zahlen: $1, \ldots, r$.
Buchstaben des Arbeitsalphabets: $1, \ldots, s$.
(Hierbei stehe 1 für $\sqcup$, 2 für 0, 3 für 1.)

Eine TM-Befehlszeile $Z = i\,j\,i'\,j', L/R/N$ werde kodiert durch das 0-1-Wort

$$\text{code(Z)}:=0^i10^j10^{i'}10^{j'}1\,0/00/000\,11$$

Die TM M mit Zeilen $Z_1, \dots, Z_l$ wird kodiert durch

$$\langle M \rangle := \mathrm{code}(Z_1) \dots \mathrm{code}(Z_l)1$$

Vereinbarung: Anfangszustand = erstgenannter Zustand, Stoppzustand = einziger Zustand, der nicht am Beginn einer Zeile auftritt.

Bemerkung:

a) Durch $\langle M \rangle$ ist M eindeutig bestimmt.

b) Es gibt einen Algorithmus, der zu $w \in \{0,1\}^*$ entscheidet, ob $w = \langle M \rangle$ für ein M.

c) In $\langle M \rangle$ kommt 111 genau einmal vor, und zwar am Ende.

Satz 3.1.2 (Unentscheidbarkeit des Wortproblems für Turingmaschinen) *Es gibt keine TM, die zur Eingabe $u \in \{0,1\}^*$ entscheidet, ob $u = \langle M \rangle w$, wobei M eine TM und w ein Wort, so dass $M : w \to$ stop.*

Kurze Formulierung: Die Sprache $Univ := \{\langle M \rangle w \mid M \text{ ist } TM, w \in \{0,1\}^*, M : w \to \text{stop}\}$ ist Turing-unentscheidbar.
Hiermit folgt die Unentscheidbarkeit des Wortproblems: Gäbe es einen Algorithmus $\mathfrak{A}$ für das Wortproblem, so auch einen für die Sprache *Univ*: Zu vorgelegtem Bitwort u bestimme, ob u die Form $\langle M \rangle w$ hat (mit b), c) der Bemerkung). Wenn nein: $u \notin Univ$, wenn ja: wende $\mathfrak{A}$ an auf $\langle M \rangle w$ und übernehme die Anwort.

Beweis der Unentscheidbarkeit von *Univ*:
Annahme: TM M_0 entscheidet *Univ*. Dann gilt

$$M_0 : \langle M \rangle w \to \begin{cases} 1 & \text{falls } M : w \to \text{stop} \\ 0 & \text{falls } M : w \to \infty \end{cases}$$

Modifiziere M_0 zu einer neuen TM M_1, die auf Eingaben $\langle M \rangle$ wie folgt arbeitet:

1. Stelle aus $\langle M \rangle$ das Wort $\langle M \rangle \langle M \rangle$ her (Kopiervorgang).
2. Arbeite wie M_0 (M_0 entscheidet hier, ob $M \langle M \rangle \to$ stop).
3. Gehe in ∞-Schleife bei M_0-Ausgabe 1, aber stoppe bei M_0-Ausgabe 0. (Implementierung: Ersetze in M_0 den Zustand q_s durch q' und füge folgende Zeilen ein: $q'\,1\,q'\,1\,N$, $q'\,0\,q_s\,0\,N$).

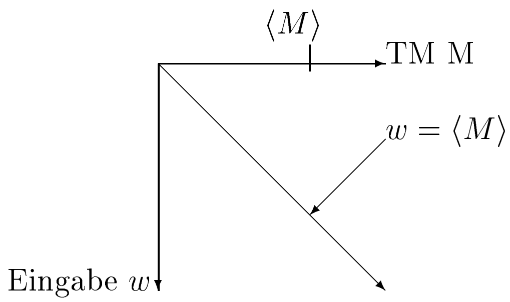
Es gilt dann für jedes Eingabewort $\langle M \rangle$:

$$M_1 : \langle M \rangle \to \text{stop} \underbrace{\iff}_{\text{nach Konstr. } M_1} M_0 : \langle M \rangle \underbrace{\langle M \rangle}_{w} \to 0 \underbrace{\iff}_{M_0 \text{ entscheidet } Univ} M : \langle M \rangle \to \infty.$$

Für $M = M_1$ ergibt sich ein Widerspruch!

□

Beweismethode: Es handelt sich um einen sogenannten *Diagonalschluss*. Im Diagramm



wird die *Diagonale* $w = \langle M \rangle$ betrachtet (Punkt 1, 2) und *modifiziert* (Punkt 3).
Ursprung: Cantors Beweis, dass die Menge der unendlichen 0-1-Folgen nicht abzählbar ist.

Annahme: Es existiert eine Liste *aller* Bitfolgen:
$\beta_0 = b_{00}b_{01}b_{02}\dots$
$\beta_1 = b_{10}b_{11}b_{12}\dots$
$\beta_2 = b_{20}\dots$
$\dots$

Betrachte die Inversion der Diagonalen: $\beta = \overline{b_{00}}\ \overline{b_{11}}\ \overline{b_{22}}\dots$.
β ist von allen β_i verschieden und fehlt somit in der Liste. Die Liste enthält nicht alle Bitfolgen, im Widerspruch zur Annahme.

Eine weitere Anwendung finden wir im folgenden überraschenden Satz:

Satz 3.1.3 (Programmierformalismen für totale Funktionen) *Sei $\mathfrak{P}$ ein Programmierformalismus mit Programmen $P_0, P_1, \dots$ so, dass:*

1. *Jedes P_i berechnet eine totale Funktion $f : \mathbb{N} \to \mathbb{N}$.*
2. *Es gibt einen Algorithmus, der zu i jeweils P_i herstellt und dann bei Eingabe von j den Ausgabewert $P_i(j)$ herstellt.*

Dann existiert eine totale, berechenbare Funktion, die durch kein P_i berechnet wird.

Programmformalismen gemäß 1. und 2. sind also notwendigerweise „unvollständig".

Beweis: Betrachte folgendes $F : \mathbb{N} \to \mathbb{N}$:

$$F(i) := P_i(i) + 1.$$

Beachte: Mit $P_i(i)$ betrachten wir die Diagonale, mit Addition von 1 modifizieren wir sie.

F ist total wegen Voraussetzung 1. F ist berechenbar wegen Voraussetzung 2.
Wir zeigen, dass F durch keines der Programme P_i berechnet wird. Annahme: Eines der P_i, etwa P_{i_0}, berechnet F. Dann gilt

$$P_{i_0}(i_0) = F(i_0) \underbrace{=}_{\text{Def. } F} P_{i_0}(i_0) + 1. \text{ Widerspruch!}$$

□

Beispielanwendungen dieses Satzes: Der Formalismus der `LOOP`-Programme liefert eine Liste $P_0, P_1, \dots$, wobei $P_i = i-$tes `LOOP`-Programm in der Folge der Wörter über Σ_{tastatur} in kanonischer Reihenfolge ist. Damit sind 1. und 2. erfüllt. Also existiert eine totale, berechenbare Funktion, die nicht `LOOP`-berechenbar ist.
Der Satz über die Unentscheidbarkeit des Wortproblems überträgt sich natürlich von TM'en auf andere Programmiersprachen, da sie ebenfalls die Simulation von TM'en gestatten.

1. Für Programme jeder gängigen (mindestens „Turing-mächtigen") Programmiersprache gilt: Es gibt keinen Algorithmus, der zu jedem Programm P und jeder Eingabe e entscheidet, ob P bei Eingabe e terminiert.

2. In vielen *Einzelfällen* ist die Entscheidung durchaus möglich. Es gibt jedoch keinen für alle Paare M, w *uniform* arbeitenden Algorithmus.

Übungen

Übung 22 (Abstrakte Form des Diagonalschlusses)

(a) Sei A eine Menge und $R \subseteq A \times A$. Für alle $a \in A$ definieren wir $R_a := \{b \in A \mid (a, b) \in R\}$.

Zeigen Sie, dass $D := \{b \in A \mid (b, b) \notin R\}$ verschieden ist von allen R_a mit $a \in A$.

(b) Sei $A = \{0, 1\}^*$ und
$R = \{(u, v) \in A \times A \mid u$ ist nicht Kodewort einer TM, die angesetzt auf v stoppt $\}$.

Zeigen Sie: Für jede entscheidbare Wortmenge L über $\{0, 1\}$ gibt es u mit $L = R_u$.

Sei D definiert wie in (a). Beschreiben Sie D möglichst einfach. Folgern Sie aus (a), dass D nicht Turing-entscheidbar ist.

Übung 23 Betrachten Sie folgende zwei Entscheidungsprobleme:

(P1) Gegeben: TM M über $\{0, 1\}$, Wort $w \in \{0, 1\}^*$, Zahl $n \geq 0$.

Frage: Stoppt M, angesetzt auf w, nach $\leq n$ Schritten?

(P2) Gegeben: TM M über $\{0, 1\}$, Wort $w \in \{0, 1\}^*$, Zahl $n \geq 0$.

Frage: Stoppt M, angesetzt auf w, nach $> n$ Schritten?

Klären Sie (mit Beweis), ob (P1) bzw. (P2) entscheidbar ist.

Übung 24 Wir betrachten diejenigen `WHILE`-Programme, die totale Funktionen $f : \mathbb{N} \to \mathbb{N}$ berechnen. Als Wörter über Σ_{tastatur} lassen sie sich (gemäß kanonischer Reihenfolge) in einer Liste $P_0, P_1, \dots$ erfassen. P_i ist also das i-te `WHILE`-Programm, welches eine totale einstellige Funktion berechnet.
Sei $F : \mathbb{N} \to \mathbb{N}$ definiert durch $F(x) = f^{(1)}_{P_x}(x) + 1$.

(a) Zeigen Sie, dass F durch kein P_i berechnet wird.

(b) Jemand sieht in (a) einen Widerspruch zur Church-Turing-These und gibt folgenden Vorschlag für einen Algorithmus zur Berechnung von F an:

Eingabe: x
Simuliere die Arbeit von P_x auf der Eingabe $(x, 0, \dots)$.
Addiere eins zum Ergebnis und stoppe.

Klären Sie (mit Beweis), ob hier ein Widerspruch zur Church-Turing-These vorliegt.

3.2 Reduktionen, weitere unentscheidbare Probleme

Ziel: Angabe weiterer unentscheidbarer Probleme.
Methode: Vergleich von Entscheidungsproblemen nach „Schwierigkeit". Dann kann man ein Problem als unentscheidbar nachweisen, indem man zeigt, dass es „mindestens so schwierig" ist wie ein schon als unentscheidbar bekanntes Problem.

Definition 3.2.1 *Ein Entscheidungsproblem $\underline{P}$ wird hier bestimmt durch:*

- *die Menge E_P der betrachteten Eingaben („Probleminstanzen"),*
- *die Menge $P \subseteq E_P$ der Eingaben, die die zu überprüfende Eigenschaft haben.*

Kurz schreiben wir $\underline{P} = (E_P, P)$.

Beispiel 1: Wortproblem $\underline{WP}$ für Turingmaschine über Σ_{bool}
E_{WP} = Menge aller Paare (Turingmaschinen über Σ_{bool}, Wort $w \in \Sigma_{bool}^*$)
WP = Menge der Paare $(M, w) \in E_{WP}$ mit $M : w \to$ stop.
Als andere, prägnantere Formulierung schreiben wir:
Gegeben: Turingmaschine M über Σ_{bool}, Wort $w \in \Sigma_{bool}^*$.
Frage: Gilt $M : w \to$ stop?

Beispiel 2: Halteproblem $\underline{H}$ für Turingmaschinen
E_H = Menge der Turingmaschinen über Σ_{bool}.
H = Menge der Turingmaschinen M mit $M : \epsilon \to$ stop.
Prägnantere Formulierung:
Gegeben: Turingmaschine M über Σ_{bool}.
Frage: Gilt $M : \epsilon \to$ stop?

Beispiel 3: Äquivalenzproblem $\underline{\ddot{A}qu}$ für Turingmaschinen über Σ_{bool}
$E_{\ddot{A}qu}$ = Menge der Paare (M_1, M_2) von Turingmaschinen über Σ_{bool}
$\ddot{A}qu$ = Menge der Paare (M_1, M_2), bei denen M_1 und M_2 die gleiche Funktion $f : \Sigma_{bool}^* - \to \Sigma_{bool}^*$ berechnen.
Prägnantere Formulierung:
Gegeben: Turingmaschinen M_1, M_2 über Σ_{bool}.
Frage: Berechnen M_1, M_2 die gleiche Funktion?

Wir führen nun den fundamentalen Begriff der Problemreduktion ein.

Definition 3.2.2 (Reduzierbarkeit) *Seien $\underline{P} = (E_P, P)$, $\underline{Q} = (E_Q, Q)$ Entscheidungsprobleme. $\underline{P}$ heißt* reduzierbar *auf $\underline{Q}$ ($\underline{P} \le \underline{Q}$), falls es eine berechenbare Funktion $f : E_P \to E_Q$ gibt derart, dass*

$$\text{für alle } x \in E_P : x \in P \Leftrightarrow f(x) \in Q$$

Intuition: „$\underline{Q}$ mindestens so schwer wie $\underline{P}$“.

Das folgende Lemma erfasst diese Intuition:

Lemma 3.2.1 (Reduktionslemma) *Es gelte $\underline{P} \leq \underline{Q}$. Dann gilt:*
$\underline{P}$ unentscheidbar $\Rightarrow$ $\underline{Q}$ unentscheidbar.
Äquivalent dazu gilt: $\underline{Q}$ entscheidbar $\Rightarrow$ $\underline{P}$ entscheidbar.

Beweis: Wir zeigen die 2. Formulierung der Behauptung
Wegen $\underline{P} \leq \underline{Q}$ gibt es eine berechenbare Funktion $f : E_P \to E_Q$ mit

$$(*) \qquad \forall\, x \in E_P: \;\; x \in P \Leftrightarrow f(x) \in Q.$$

Sei A_f ein Algorithmus, der f berechnet.
Da $\underline{Q}$ entscheidbar ist existiert ein Algorithmus A_Q, der zu $y \in E_Q$ entscheidet, ob $y \in Q$.
Gesucht ist ein Algorithmus A_P, der $\underline{P}$ entscheidet. Wir definieren A_P so:

$$A_P \left| \begin{array}{l} \text{Bei Eingabe } x \in E_P \text{ wende } A_f \text{ an, erhalte } f(x) \in E_Q \\ \text{Auf } f(x) \text{ wende } A_Q \text{ an, erhalte Antwort, ob } f(x) \in Q \\ \text{Ausgabe sei diese Antwort von } A_Q. \text{ (Diese Aussage ist korrekt wegen } (*)\text{).} \end{array} \right.$$

□

Als Rückblick noch einmal die Idee zu $\underline{P} \leq \underline{Q}$: Die Frage „$x \in P$?“ wird abgewälzt auf die Frage „$f(x) \in Q$?“. Man kann also die Frage „$x \in P$?“ via f als Frage über Q kodieren.

Anwendungen:

Satz 3.2.1 *Das Halteproblem $\underline{H}$ für Turingmaschinen ist unentscheidbar.*

Beweis: Nach Reduktionslemma genügt: $\underline{WP} \leq \underline{H}$.

Finde also: $f : E_{WP} \to E_H$ berechenbar mit

$$M : w \to \mathit{stop} \;\Leftrightarrow M' : \epsilon \to \mathit{stop}.$$

Konstruiere M' aus (M, w) wie folgt:
M' schreibt zunächst w auf das leere Band, geht auf erstes Feld zurück und arbeitet dann wie M.
Es ist klar, dass $f : (M, w) \mapsto M'$ berechenbar ist. Nach Konstruktion von M' gilt auch: $M : w \to \mathit{stop} \;\Leftrightarrow\; M' : \epsilon \to \mathit{stop}$, d.h. $(M, w) \in WP$ gdw. $M' \in H$.

□

Satz 3.2.2 *Das Äquivalenzproblem $\underline{\ddot{A}qu}$ für Turingmaschinen über Σ_{bool} ist unentscheidbar.*

Beweis: Es genügt zu zeigen: $\underline{H} \leq \underline{\ddot{A}qu}$ (wegen Reduktionslemma und Satz 3.2.1).

Zu finden ist also $f : E_H \to E_{\ddot{A}qu}$ mit
$M \mapsto (M_1, M_2)$

$$\underset{(M \in H)}{M : \epsilon \to stop} \quad \Leftrightarrow \quad \underset{((M_1,M_2) \in \ddot{A}qu)}{M_1, M_2 \text{ berechnen gleiche Funktion}}$$

Konstruiere M_1 aus M wie folgt:
Bei Eingabe w löscht M_1 das Wort w, arbeitet dann wie M (auf ϵ !), und falls M dann stoppt, wird ein ␣ auf das aktuelle Arbeitsfeld gedruckt („Ausgabe ϵ“).

Also gilt für jedes w: $M_1 : w \to \epsilon$, falls $M : \epsilon \to stop$
$M_1 : w \to \infty$, falls $M : \epsilon \to \infty$.

Wir halten fest: M_1 berechnet die Konstante ϵ, falls $M : \epsilon \to stop$.
M_1 berechnet die überall undefinierte Funktion, falls $M : \epsilon \to \infty$.

Konstruiere M_2 (unabhängig von M) so: Bei Eingabe w drucke ␣ und stoppe.
M_2 berechnet die Konstante ϵ.
Insgesamt erhalten wir: Die Transformation $M \mapsto (M_1, M_2)$ ist berechenbar und es gilt $M : \epsilon \to stop \;\Leftrightarrow\; M_1, M_2$ berechnen dieselbe Funktion.

□

Wir halten fest: Es gilt insbesondere für höhere Programmiersprachen (mit denen man Turingmaschinen simulieren kann):
Es gibt keinen Algorithmus, der die Äquivalenz (bzgl. Eingabe-Ausgaberelation) beliebiger Programme entscheidet.

Dass das Äquivalenzproblem unentscheidbar ist, hat auch für die fundamentale Frage der *Vereinfachung von Programmen* unangenehme Konsequenzen. Wenn man nach irgendeinem Kriterium „das bestmögliche Programm“ zur Berechnung einer Funktion f sucht (z.B. mit kleinstem Wachstum an Laufzeit, bezogen auf die Argumente), so könnte man ein Transformationsverfahren anstreben, das aus einem vorgelegten Programm ein äquivalentes optimales Programm in diesem Sinne herstellt. Dieser Ansatz kann nicht gelingen, wenn man auf einer algorithmisch durchführbaren Optimierung besteht und wenn es zu jeder Funktion f nur ein eindeutig bestimmtes „optimales“ Programm zur Berechnung von f geben soll. Denn die Anwendung des Transformationsverfahrens auf zwei gegebene Programme würde sofort das Äquivalenzproblem lösen; man hat nur zu prüfen, ob die beiden hergestellten „optimalen“ Programme übereinstimmen. *Es gibt also für* `WHILE`*-Programme* (und damit für Programme in höheren Programmiersprachen) in diesem Sinne *keine Möglichkeit, „beste Programme“ herzustellen.*

Wir verschärfen jetzt die Unentscheidbarkeitsresultate durch eine sehr weitgefasste Aussage: *„Jedes interessante Problem über Turingmaschinen ist unentscheidbar“*. Als Eingabemenge verwenden wir also E_{TM} = Menge der Turingmaschinen über Σ_{bool}.

Die „interessanten“ Probleme werden in der folgenden Definition beschrieben:

Definition 3.2.3 *Ein Entscheidungsproblem* $\underline{P} = (E_{TM}, P)$ *über Turingmaschinen heißt*

- semantisch, *wenn die Mitgliedschaft in* P *nur vom Eingabe-/Ausgabe-Verhalten der Turingmaschine abhängt, d.h.: Berechnen* M_1, M_2 *dieselbe Funktion* $f : \Sigma_{bool}^* - \rightarrow \Sigma_{bool}^*$, *dann gilt:* $M_1 \in P \Leftrightarrow M_2 \in P$.
- nicht-trivial, *wenn es Turingmaschinen* $M \in P$ *und Turingmaschinen* $M \notin P$ *gibt.*

Satz 3.2.3 (von Rice) *Jedes semantische, nicht-triviale Entscheidungsproblem über Turingmaschinen ist unentscheidbar.*

Beweis: Betrachte $\underline{P} = (E_{TM}, P)$ semantisch, nicht trivial.
$f_\perp$ sei die überall undefinierte Funktion auf $\{0,1\}^*$, d.h. definiert durch $f_\perp(w) = \perp$ für $w \in \{0,1\}^*$.
1. Fall: Die TM'en, die $f_\perp$ berechnen, gehören alle nicht zu P.
2. Fall: Die TM'en, die $f_\perp$ berechnen, gehören alle zu P.
(Da P semantisch ist, muss einer der beiden Fälle zutreffen!)

Wir betrachten den 1. Fall und zeigen $\underline{H} \leq \underline{P}$. (Dann sind wir wegen der Unentscheidbarkeit des Halteproblems $\underline{\mathrm{H}}$ fertig).

Finde also eine Transformation $F : E_{TM} \rightarrow E_{TM}, M \mapsto M'$ berechenbar, mit $M \in H$ (d.h. $M : \epsilon \rightarrow \mathit{stop}) \Leftrightarrow M' \in P$.
Da P nicht-trivial und semantisch ist und Fall 1 gilt, wähle TM $M_h \in P$, die eine Funktion $h \neq f_\perp$ berechnet. Nun konstruieren wir M' aus M so:

M' | Bei Eingabe w arbeite wie M auf ϵ (und bewahre w auf freiem Bandbereich auf).
Wenn M stoppt, bringe w an den Bandanfang und arbeite dann wie M_h auf w.

Nach dieser Festlegung ist die Transformation $M \mapsto M'$ berechenbar. Wir haben zu prüfen:

$$(*) \qquad M : \epsilon \rightarrow \mathit{stop} \Leftrightarrow M' \in P \qquad .$$

Falls $M : \epsilon \rightarrow$ stop, arbeitet M' wie M_h, berechnet also h. Da $M_h \in P$ und P semantisch ist und M', M_h dieselbe Funktion berechnen, gilt $M' \in P$.
Falls $M : \epsilon \rightarrow \infty$, bleibt M' in der 1. Phase „hängen“, berechnet somit $f_\perp$.
Wegen Fall 1 gilt also $M' \notin P$. Damit ist der erste Fall erledigt. Der 2. Fall geht völlig analog.

□

Übungen

Übung 25 Sei $\underline{H}_{111}$ das folgende Entscheidungsproblem:

Gegeben: TM M über $\{0,1\}$.

Frage: Akzeptiert M das Wort 111?

Zeigen Sie: $\underline{H}_{111} \leq \underline{H}$ und $\underline{H} \leq \underline{H}_{111}$. (Hierbei ist $\underline{H}$ das Halteproblem im Sinne der Vorlesung.)

Übung 26 Eine TM M über Alphabet $\{0, 1, \$\}$ heiße *eingabebeschränkt*, wenn sie beim Lesen eines \$ stets erstens ein \$ schreibt und zweitens nicht nach rechts geht. Eine eingabebeschränkte TM kann also bei Eingabe $w\$$ (für ein Wort w der Länge n über $\{0, 1\}$) nur die ersten $n + 1$ Felder besuchen.
Zeigen Sie, dass folgendes Problem entscheidbar ist:

Gegeben: Eingabebeschränkte TM M und $w \in \{0, 1\}^*$.

Frage: Akzeptiert M die Eingabe $w\$$?

Hinweis: Wieviele verschiedene Konfigurationen kann es geben?

Übung 27 Ist M eine TM über $\{0, 1\}$, so sei

$$\mathrm{Def}(M) = \{w \in \{0, 1\}^* \mid M : w \to \mathrm{stop}\}.$$

Zeigen Sie durch eine geeignete Reduktion, dass folgendes Problem unentscheidbar ist:

Gegeben: TMen M_1, M_2 über $\{0, 1\}$.

Frage: Gilt $\mathrm{Def}(M_1) \subseteq \mathrm{Def}(M_2)$?

Übung 28 Ein $\mathtt{GOTO}'_1$-*Programm* ist $\mathtt{GOTO}'$-Programm P mit $maxind(P) = 1$. Zeigen Sie, dass folgendes Problem entscheidbar ist:

Gegeben: $\mathtt{GOTO}'_1$-Programm P.

Frage: Ist $f_P^{(1)}(0)$ definiert?

Übung 29 Wir betrachten Entscheidungsprobleme der Form $\underline{P} = (\mathbb{N}, P)$ mit Eingabemenge $\mathbb{N}$. Für $\underline{P_1} = (\mathbb{N}, P_1)$ und $\underline{P_2} = (\mathbb{N}, P_2)$ sei $\underline{P_1} \oplus \underline{P_2} = (\mathbb{N}, P_1 \oplus P_2)$ mit

$$P_1 \oplus P_2 := \{2n \mid n \in P_1\} \cup \{2n + 1 \mid n \in P_2\}.$$

Zeigen Sie:

(a) $\underline{P_1} \leq \underline{P_1} \oplus \underline{P_2}$ und $\underline{P_2} \leq \underline{P_1} \oplus \underline{P_2}$.

(b) Für alle Entscheidungsprobleme $\underline{Q}$ mit Eingabemenge $\mathbb{N}$ gilt: Wenn $\underline{P_1} \leq \underline{Q}$ und $\underline{P_2} \leq \underline{Q}$, so $\underline{P_1} \oplus \underline{P_2} \leq \underline{Q}$.

Übung 30 Zeigen Sie, dass folgendes Problem unentscheidbar ist:

Gegeben: TM M über Eingabealphabet $\{0, 1\}$.

Frage: Überdruckt M, angesetzt auf das leere Band, einmal ein Symbol $\neq \sqcup$ durch $\sqcup$?

Übung 31 Seien $L, M \subseteq \{0, 1\}^*$. Zeigen Sie: Ist M aufzählbar und $L \leq M$, so ist L aufzählbar.

Übung 32 Sei $L \subseteq \{0,1\}^*$ aufzählbar mit $L \leq \{0,1\}^* \setminus L$.
Zeigen Sie: L ist entscheidbar.

Übung 33 Seien TOTAL und INFIN die Menge aller TMen über $\{0,1\}$, welche für alle (bzw. für unendlich viele) Eingaben aus $\{0,1\}^*$ terminieren.
Zeigen Sie: $\underline{\text{INFIN}} \leq \underline{\text{TOTAL}}$.
Hinweis: dove-tailing.

Übung 34 (a) Für jede nichtleere aufzählbare Sprache $L \subseteq \{0,1\}^*$ gibt es eine totale, rekursive Funktion $f : \{0,1\}^* \to \{0,1\}^*$ mit $\text{Bild}(f) = L$.

(b) $\underline{\text{TOTAL}}$ ist nicht aufzählbar.
Hinweis zu (b): Verwenden Sie das f aus Teil (a) für einen Diagonalschluss.

3.3 Dominoproblem und Post'sches Korrespondenzproblem

Bisher haben wir nur gezeigt, dass bestimmte Probleme, die die Gesamtheit der `WHILE`-Programme betreffen, unentscheidbar sind. Man kann einwenden, dass dies nicht überraschend ist, wenn man die Church-Turing-These unterstellt: Denn ein in Frage kommendes Entscheidungsverfahren wird durch ein `WHILE`-Programm erfasst, und dieses müsste dann über alle `WHILE`-Programme, also auch über sich selbst, Auskunft geben. In dieser Situation liegt es nahe, dass Schwierigkeiten auftreten (und in der Tat wird ja solch ein Selbstbezug im Diagonalschluß des vorangehenden Abschnitts verwendet). Wir wollen nun ein Problem als unentscheidbar nachweisen, das nicht mit Programmen zu tun hat, sondern in einer ganz elementaren kombinatorischen Fragestellung besteht. Es handelt sich um das sog. „Dominoproblem", bei dem es darum geht, mit gefärbten quadratischen Plättchen die euklidische Ebene zu pflastern (zu „parkettieren"). Wir begnügen uns mit der Pflasterung des 4. Quadranten („nach rechts und unten"). Ein Domino sei ein quadratisches (gerichtetes) Plättchen mit je einer „Farbe" auf jeder der vier Kanten. Ein Dominotyp sei ein Quadrupel von Farben (etwa in der Reihenfolge „Nord-, Ost-, Süd-, West-Farbe"). Wir denken uns Farben gegeben durch Zahlen oder Zeichenreihen über einem geeigneten Alphabet (etwa dem ASCII-Zeichensatz). Ein Dominospiel ist eine endliche Folge $D = (D_0, \ldots, D_k)$ von Dominotypen. Das Spiel heiße gut, wenn man mit einem unbegrenzten Vorrat von Dominos der Typen $D_0, \ldots, D_k$ den 4. Quadranten der Ebene so belegen kann, dass ein Domino vom Typ D_0 in der linken oberen Ecke liegt und die Farben von Kanten aneinandergrenzender Dominos übereinstimmen.

Beispiel 1: Dominospiel D_0 enthalte die Dominotypen $d_0 = \boxed{\begin{array}{ccc} & 1 & \\ 3 & & 3 \\ & 2 & \end{array}}$ $\quad d_1 = \boxed{\begin{array}{ccc} & 2 & \\ 3 & & 3 \\ & 1 & \end{array}}$

Es gibt eine Parkettierung des 4. Quadranten mit d_0 an Eckposition: D_0 ist „gut".
Eine mögliche Parkettierung ist:

$$\begin{array}{|llll}
\hline
d_0 & d_0 & d_0 & \dots \\
d_1 & d_1 & d_1 & \dots \\
d_0 & d_0 & \dots & \\
\vdots & & &
\end{array}$$

Beispiel 2: D_1 enthalte die Dominotypen $d_0 = \begin{array}{|ccc|}\hline & 1 & \\ 3 & & 3 \\ & 2 & \\ \hline\end{array}$ $d_1 = \begin{array}{|ccc|}\hline & 2 & \\ 1 & & 1 \\ & 3 & \\ \hline\end{array}$ $d_2 = \begin{array}{|ccc|}\hline & 3 & \\ 3 & & 2 \\ & 1 & \\ \hline\end{array}$

D_1 lässt keine Parkettierung des 4. Quadranten mit d_0 an Eckposition zu: D_1 ist „nicht gut“.
Eine Parkettierung muss offenbar wie folgt beginnen:

$$\begin{array}{|lll}
\hline
d_0 & d_0 & \dots \\
d_1 & d_1 & \dots \\
d_2 & * & \\
\vdots & &
\end{array}$$

Bei $*$ lässt sich die Parkettierung aber nicht fortsetzen!

Definition 3.3.1 *Ein* Dominotyp *ist ein Quadrupel $d = (m_N, m_O, m_S, m_W)$ von natürlichen Zahlen („Farben“).*

$$\begin{array}{|ccc|}\hline & m_N & \\ m_W & & m_O \\ & m_S & \\ \hline\end{array}$$

$$\textit{Setze } N(d) := m_N, O(d) := m_O, S(d) := m_S, W(d) := m_W$$

Ein Dominospiel *ist eine Folge $D = (d_0, \dots, d_k)$ von Dominotypen.*

Eine Parkettierung *mit D ist eine Abbildung $\pi : \mathbb{N} \times \mathbb{N} \to \{d_0, \dots, d_k\}$ mit:*

- $\pi(0,0) = d_0$ *(Eckdominobedingung)*
- $O(\pi(i,j)) = W(\pi(i,j+1))$, *für alle* $(i,j) \in \mathbb{N} \times \mathbb{N}$
- $S(\pi(i,j)) = N(\pi(i+1,j))$, *für alle* $(i,j) \in \mathbb{N} \times \mathbb{N}$.

D heiße *gut*, falls es eine Parkettierung mit D gibt.

Das **Eckdominoproblem** ist das Entscheidungsproblem $\underline{D} = (E_D, D)$, wobei E_D die Menge der Dominospiele und D die Menge der nicht guten Dominospiele ist.
Die prägnante Formulierung des Eckdominoproblems lautet also:

Gegeben: Dominospiel D

Frage: Ist D nicht gut ?

Satz 3.3.1 *Das Eckdominoproblem ist unentscheidbar.*

Beweis: Wir reduzieren das Halteproblem für TM auf das Eckdominoproblem, zeigen also $\underline{H} \leq \underline{D}$. Zu finden ist eine berechenbare Funktion $F : E_H \to E_D$, d.h. eine Transformation $M \mapsto D_M$ von Turingmaschinen in Dominospiele so, dass gilt: $M : \epsilon \to stop \Leftrightarrow D_M$ nicht gut.

Idee: Konstruiere D_M so, dass es höchstens eine Parkettierung gibt und zwar so, dass Zeile für Zeile die M-Konfigurationen beginnend mit der Anfangskonfiguration auf leerem Band kodiert sind. Dann werden wir erhalten, dass diese Parkettierung abbricht gdw. M auf ϵ terminiert. Damit wird wie gewünscht gelten: $M : \epsilon \to stop \Leftrightarrow D_M$ nicht gut.

□

Beispiel zur Transformation $M \mapsto D_M$:
M habe die Befehle (M über $\{0, 1\}$):

$q_0 \sqcup q_1\ 0\ R$

$q_1 \sqcup q_1\ 1\ N$

$q_1\ 1\ q_0\ 1\ L$

sonstige Zeilen: $q\ a\ q_s\ a\ N$, für alle $q \in \{q_0, q_1\}, a \in \{0, 1\}$

Wir verfolgen die Konfigurationsfolge mit Start auf leerem Band.

$q_0 \sqcup \sqcup \sqcup \sqcup \ \cdots$

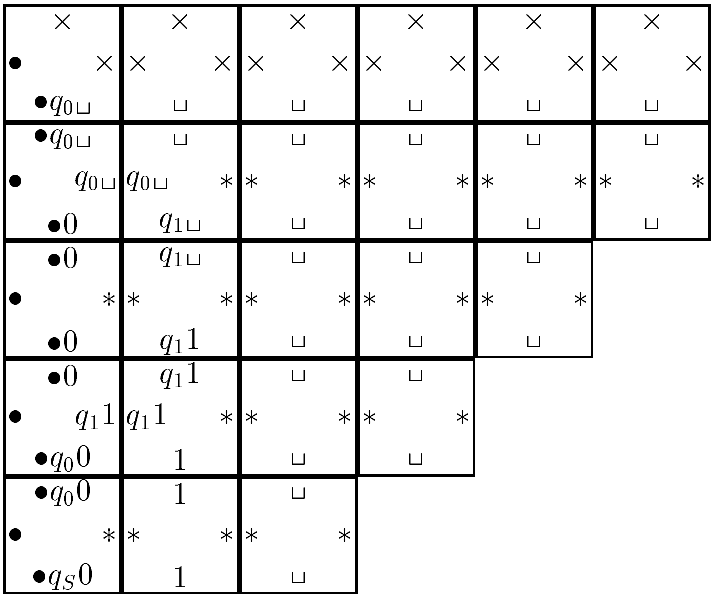
$0\ q_1 \sqcup \sqcup \sqcup \ \cdots$

$0\ q_1\ 1 \sqcup \sqcup \ \cdots$

$q_0\ 0\ 1 \sqcup \sqcup \ \cdots$

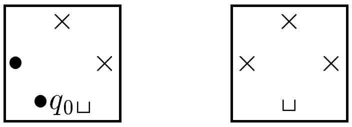
$q_s\ 0\ 1 \sqcup \sqcup \ \cdots$

Die gewünschte Parkettierung (nur für die ersten Zeilen) lautet somit:

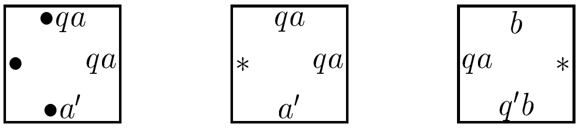


Die „unteren Farbfolgen" kodieren also sukzessiv die TM-Konfigurationen.
Allgemeine Definition von D_M zur Turingmaschine $M = (Q, \Sigma, \Gamma, q_0, \delta, q_s)$

Top-Dominos:

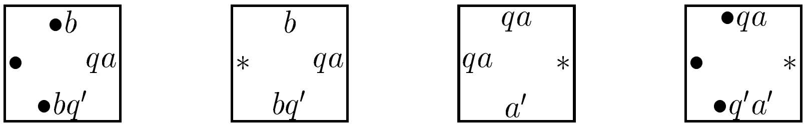


Falls Befehlszeile $q\ a\ q'\ a'\ R$ in M:



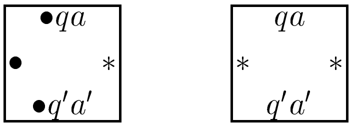
für jedes $b \in \Gamma$

Falls Befehlszeile $q\ a\ q'\ a'\ L$ in M:

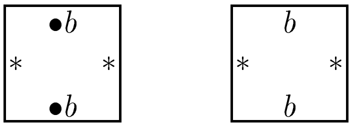


für jedes $b \in \Gamma$

Falls Befehlszeile $q\ a\ q'\ a'\ N$ in M:



Außerdem:



für jedes $b \in \Gamma$

Offenbar ist diese Transformation $F : M \mapsto D_M$ berechenbar.

Wir haben zu prüfen:

$$M : \epsilon \to stop \Leftrightarrow D_M \text{ nicht gut.} \qquad (*)$$

Hierzu genügt offenbar folgendes Lemma:

Lemma 3.3.1 *Für $i \geq 0$ läuft M angesetzt auf das leere Band i Schritte, etwa bis zur Konfiguration κ_i, gdw. D_M gestattet eine einzige Parkettierung der ersten Zeilen $(0, \ldots, i)$ und in dieser Parkettierung ist die untere Farbfolge von Zeile i das Wort $\kappa_i \sqcup\sqcup\sqcup \ldots$*

Beweis: Durch Induktion über i:

$i = 0$: Klar, wegen Konstruktion von D_M (Rückgriff auf die ersten beiden Dominotypen).

Induktionsschritt: M laufe $i+1$ Schritte. Nach i Schritten sei κ_i erreicht. Nach Induktionsvoraussetzung erlaubt D_M genau eine Parkettierung auf Zeilen $0, \ldots, i$, und zwar mit unterer Farbfolge $\kappa_i \sqcup\sqcup\sqcup \ldots$.
Gemäß Definition von D_M (ausgehend von Farbe qa) ist genau eine Fortsetzung auf Zeile $i+1$ möglich (Fallunterscheidung nach den Befehlsoptionen $L/R/N$, und den Fällen Nicht-Rand/Rand). Neue Farbfolge liefert die Folgekonfiguration von κ_i.

$\square$

Wir zeigen nun $(*)$. Hierzu genügt:

(a) $M : \epsilon \to \infty \Rightarrow D_M$ gut

(b) $M : \epsilon \to \mathit{stop} \Rightarrow D_M$ nicht gut

zu (a): Nach Lemma sind für jedes i die Zeilen $(0, \ldots, i)$ eindeutig parkettierbar, insgesamt gibt es also eine eindeutige Parkettierung, die alle Zeilen parkettiert.

zu (b): Nach i Schritten erreiche M die Stoppkonfiguration mit $q_s a$ in unterer Farbfolge von Zeile i. Nach Konstruktion von D_M fehlt $q_s a$ als obere Farbe. Somit ist Zeile $i+1$ nicht parkettierbar, also ist D_M nicht gut.

$\square$

Wir stellen ohne Beweis ein weiteres wichtiges unentscheidbares Problem vor:

Das Post'sche Korrespondenzproblem (PCP)

Gegeben: Zwei Listen $(u_1, \ldots, u_m), (v_1, \ldots, v_m)$ von (gleich vielen) Wörtern.

Frage: Gibt es eine Indexfolge $i_1, \ldots, i_n (n \geq 1)$ so, dass $u_{i_1} \ldots u_{i_n} = v_{i_1} \ldots v_{i_n}$?

Satz 3.3.2 *PCP ist unentscheidbar.*

Rückblick zum Begriff der Entscheidbarkeit

Das Ergebnis etwa über die Unentscheidbarkeit des Dominoproblems bedeutet natürlich nicht, dass es für ein betrachtetes Dominospiel unmöglich ist, die Parkettierungsfrage zu klären. Für konkret gegebene Dominospiele ist häufig durch eine Analyse des Einzelfalls klar, ob nun eine Parkettierung existiert oder nicht.
Analoges gilt für konkret vorgelegte Instanzen des PCP-Problems. Das Unentscheidbarkeitsresultat besagt nur, dass man nicht mit Hilfe eines einzigen Algorithmus für *alle* Fragestellungen uniform die richtige Antwort bekommen kann. Damit ist man bei einem unentscheidbaren Problem auf die Analyse der Einzelfälle zurückgeworfen, und jeder dieser Fälle für sich mag spezielle Ideen zu seiner Lösung erfordern.
Zuweilen wird in Grundlagendiskussionen auch für solche Einzelfragen behauptet, sie seien „nicht entscheidbar“. Z.B. wird die mengentheoretische Aussage der Cantor'schen „Kontinuumshypothese“ (dass eine unendliche Menge natürlicher Zahlen immer gleichmächtig

zu $\mathbb{N}$ oder zu $\mathbb{R}$ sei) als „unentscheidbar“ bezeichnet. Damit ist dann eine andere Aussage gemeint, nämlich dass in einem bestimmten formalen System der Mathematik weder die betrachtete Aussage noch ihr Negat formal herleitbar ist. Bei der Kontinuumshypothese trifft dieser Defekt in Bezug auf das übliche Axiomensystem der Mengenlehre zu. Es handelt sich also bei dieser Verwendung des Begriffs „entscheidbar“ um die Frage nach der Existenz eines formalen Beweises einer Einzelaussage und nicht um die Frage nach der Existenz eines Algorithmus zur Beantwortung einer Klasse von Fragen. Wir beziehen uns im folgenden immer auf letztere (algorithmische) Fragestellung.
Schränkt man unentscheidbare Probleme geeignet ein, so kann sich die Entscheidbarkeit ergeben. Das zeigt sich z.B. beim Dominoproblem. Wenn wir uns etwa beim Dominoproblem nur für die Parkettierung einer einzigen Zeile interessieren, kann man ein Entscheidungsverfahren angeben (vgl. Übung).

Die folgende Bemerkung macht noch einmal klar, dass der Begriff der Unentscheidbarkeit nur für unendliche Klassen von Probleminstanzen vernünftig ist.
Bemerkung: Ein Entscheidungsproblem $\underline{P} = (E_P, P)$ mit endlicher Eingabemenge E_P ist entscheidbar.

Beweis: Sei $E_P = \{e_1, \ldots, e_n\}$. Ein Algorithmus hat die Form:

$$\text{Zu Eingabe } e \quad \left.\begin{array}{ll} \text{ist } e = e_1 & \text{so terminiere mit Ausgabe ja/nein} \\ \vdots & \qquad\qquad\qquad\vdots \\ \text{ist } e = e_1 & \text{so terminiere mit Ausgabe ja/nein} \end{array}\right\} \; 2^n \text{ Möglichkeiten}$$

Einer dieser 2^n Algorithmen entscheidet $\underline{P}$. Allerdings lässt diese Existenzaussage offen, welcher der 2^n Algorithmen der richtige ist!

$\square$

Übungen

Übung 35 Zeigen Sie, dass das folgende „Streifen-Dominoproblem” entscheidbar ist:

Gegeben: Dominospiel $D = (d_0, \ldots, d_k)$.

Frage: Lässt sich mit D (beginnend mit d_0) der erste Streifen des 4. Quadranten parkettieren?

3.4 Ausblick in die Rekursionstheorie

Wir schließen unsere Betrachtungen zur Unentscheidbarkeit mit kurzen Andeutungen über die weitreichenden Ergebnisse der *Rekursionstheorie* ab. Näheres in vertiefenden Vorlesungen oder in den Lehrbüchern von Börger, Odifreddi, Rogers (vgl. Literaturliste).
Erinnerung: Die Sprache *Univ* war definiert durch

$$Univ = \{code(M)w \mid M \text{ ist } TM \text{ über } \{0,1\}, w \in \{0,1\}^*, M : w \to \text{stop}\}.$$

Gezeigt: *Univ* ist nicht entscheidbar. Turing zeigte jedoch:

Satz 3.4.1 (Turing) *Univ ist semi-entscheidbar.*

Beweis durch Angabe einer semi-entscheidenden TM M_0, die so arbeitet:

1. Bei Eingabe v wird geprüft, ob v die Gestalt $code(M)w$ hat. (Wenn nein, terminiere nicht.)
2. Falls $v = code(M)w$, simuliere auf Bandabschnitt nach $code(M)$ die Arbeit von M auf w. Bei Termination gebe 1 aus und stoppe.

□

Ein solches M_0 heißt *universelle Turingmaschine*. Sie ist das theoretische Modell der *programmierbaren Rechenmaschine*, die als Eingabe sowohl ein Programm (hier $code(M)$) als auch Daten dafür (hier w) enthält.
Erinnerung (Satz 1.2.4): Eine Sprache L ist semi-entscheidbar $\Leftrightarrow$ L ist aufzählbar. Bei Bezug auf TM'en sagt man statt „Turing-aufzählbar“ auch einfach „rekursiv-aufzählbar“.

Bemerkung: Es gibt nicht-aufzählbare Sprachen, z.B. $\{0,1\}^* \setminus Univ$.

Beweis: Annahme: $\{0,1\}^* \setminus Univ$ sei aufzählbar. Nach dem Satz von Turing gilt: $Univ$ ist aufzählbar. Nach Satz 1.2.2 ist somit $Univ$ entscheidbar im Widerspruch zur Unentscheidbarkeit von $Univ$ (Satz 3.4.1).

□

Die aufzählbaren Sprachen lassen sich wie folgt „im Entscheidbaren verankern“:

Satz 3.4.2 *$L \subseteq \{0,1\}^*$ aufzählbar $\Leftrightarrow$ es gibt eine entscheidbare Relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ mit $u \in L \Leftrightarrow \exists v (u,v) \in R$.*

Beweis: „$\Rightarrow$“ Der Algorithmus $\mathfrak{A}$ zähle die L-Wörter auf. Definiere: $R := \{(u,v) |$ Nach $|v|$ Schritten liefert $\mathfrak{A}$ Ausgabe $u\}$.
Offensichtlich ist R entscheidbar, und es gilt: $u \in L \Leftrightarrow \exists v (u,v) \in R$.

„$\Leftarrow$“ Konstruiere Aufzählungsalgorithmus für L: Gehe alle Wortpaare (u,v) systematisch durch (etwa nach Summe der Längen von u,v) und überprüfe mit Entscheidungsverfahren für R jeweils, ob $(u,v) \in R$. Wenn ja, gebe jeweils u aus.

□

Wendet man auf unentscheidbare Relationen nicht nur einen Existenzquantor an, sondern abwechselnd $\forall$- und $\exists$-Quantoren, so erhält man die sogenannte „arithmetische Hierachie“:

Definition 3.4.1 *L heißt Σ_n-Sprache gdw. es gibt eine entscheidbare Relation $R \subseteq (\{0,1\}^*)^{n+1}$ mit $u \in L \Leftrightarrow \exists v_1 \forall v_2 \exists v_3 \ldots \exists/\forall v_n (u, v_1, \ldots, v_n) \in R$.*
Also: L aufzählbar $\Leftrightarrow$ L ist Σ_1-Sprache.

In der Rekursionstheorie zeigt man, dass man für wachsendes n mit den Σ_n-Sprachen jeweils immer größere Sprachklassen bekommt.
Zurück zu Σ_1. Hier kann man zeigen, dass die Sprache $Univ$ eine Sonderrolle spielt:

Satz 3.4.3 *Die Sprache $Univ$ ist aufzählbar und für jedes aufzählbare L gilt: $L \leq Univ$.*

Beweis: Sei L aufzählbar, also semi-entscheidbar. M semi-entscheide L.
Definiere Reduktion $\underbrace{f : u \mapsto u'}_{\text{berechenbar}}$ mit $u \in L \Leftrightarrow u' \in Univ$. Dies gilt mit der simplen Festlegung $u' := code(M)u$.

□

Man sagt $Univ$ ist *vollständig* bezüglich $\leq$ in der Klasse der aufzählbaren Sprachen.

Abschließend ein fundamentales, aber auch merkwürdiges Ergebnis. Wir ordnen hierzu jedem Wort $w \in \Sigma_{bool}^*$ eine TM M_w zu:
Zu $w \in \{0,1\}^*$ sei M_w

- die TM M mit $code(M) = w$, falls w Kodewort ist.
- die TM mit den Zeilen $q_0 a q_0 a N$ (die nie terminiert), falls w kein TM-Kodewort ist.

Satz 3.4.4 (Rekursionstheorem) *Sei $s : \{0,1\}^* \to \{0,1\}^*$ berechenbar und total. Dann existiert ein Wort u derart, dass M_u und $M_{s(u)}$ dieselbe Funktion berechnen.*

Dieses Resultat kann man auch formulieren als die Aussage *„Programmierte Programmsabotage ist nie perfekt“*. Wir stellen uns einen Saboteur vor, der TM-Codes u in Codes $s(u)$ so umwandeln will, dass die berechneten Funktionen sich ändern. Er will also erreichen, dass M_u und $M_{s(u)}$ verschiedene Funktionen berechnen. Der Satz sagt, dass das nie voll gelingen kann, wenn die „Sabotagetransformation“ S algorithmisch durchgeführt wird (also berechenbar ist): Es wird dann immer einen Code u geben, so dass der sabotierte Code $s(u)$ die *gleiche* Funktion wie u berechnet. Im folgenden schreiben wir einfach M_w statt f_{M_w}, also für die Funktion, die die TM M mit Code w berechnet.
Beweis: Zu $w \in \{0,1\}^*$ arbeite die TM $M[w]$ wie folgt:

1. Berechne $M_w(w)$
2. Falls $M_w(w)$ definiert ist, wende $M_{M_w(w)}$ auf die Eingabe x an.

Die Transformation $g : w \to code(M[w])$ ist offenbar berechenbar und total.
Definiere h durch $h(w) = s(g(w))$. Dann ist auch h berechenbar und total. Sei v Kodewort einer TM, die h berechnet. Es gilt dann $M_v(w) = h(w) = s(g(w))$ für jedes w. Setze $u := g(v)$. Für dieses u prüfen wir die Behauptung: $M_u(x) = M_{s(u)}(x)$ für alle x.

$$M_u(x) = M_{g(v)}(x) \underbrace{=}_{\text{Def. } g} M_{code(M[v])}(x) = M[v](x) \underbrace{=}_{\substack{\text{Def. } M[v] \\ M_v \text{ total}}} M_{M_v(v)}(x) = M_{s(g(v))}(x) = M_{s(u)}(x)$$

□

Kapitel 4

Zeitkomplexität

4.1 Einführung: Zeitbeschränkte Turingmaschinen

Die bisherigen Überlegungen dienten der Klärung der Begriffe „Berechenbarkeit“ und „Entscheidbarkeit“ als Bestimmung der Grenzen für die Lösung algorithmischer Probleme.
Beobachtung: Obwohl ein Problem im Prinzip lösbar ist, ist es u.U. in der Praxis nicht lösbar, weil ein Algorithmus zuviel Zeit, Platz etc. benötigt.
Ziel: Genauere Analyse der Ressourcen, z.B. der Zeit, die für die Lösung eines Problems nötig sind.

Definition 4.1.1 („worst-case“ Zeitkomplexität) *Sei M eine k-Band-Turingmaschine, die auf jeder Eingabe stoppt. Die Laufzeit oder Zeitkomplexität in M ist die Funktion*

$$f : \mathbb{N} \to \mathbb{N},\ n \mapsto \max\{m \mid \textit{ es existiert Eingabe } w \textit{ der Länge } n \textit{ so, dass } M \textit{ auf } w\ m \textit{ Schritte läuft}\}$$

Wenn f die Laufzeit von M ist, so bezeichnen wir M als $f(n)$-Zeit-Turingmaschine.
Falls $f(n)$ $O(g(n))$ ist, so sagen wir, dass M eine $O(g(n))$-zeitbeschränkte Turingmaschine ist.

Erinnerung: Seien $f, g : \mathbb{N} \to \mathbb{N}$ total, so besagt „$f(n)$ ist $O(g(n))$“ , dass es ein $c > 0$ gibt so, dass $n_0 > 0$ existiert für alle $n \geq 0$ gilt $f(n) \leq c \cdot g(n)$

Beispiel: Wir betrachten die Sprache $L = \{0^k 1^k \mid k \geq 0\}$ und untersuchen die Zeitkomplexität von TM'en, die diese Sprache entscheiden. Betrachte die Turingmaschine M_1, welche wie folgt arbeitet:

$\boxed{0}$	0	…	0	1	…	1	1	␣	…	……

Phase 1: Laufe einmal über die Eingabe, verwerfe, falls Infix 10 auftaucht.
Phase 2: Laufe einmal über das Band und prüfe, ob noch 0en und 1en vorhanden sind.
Falls noch 0en <u>und</u> 1en: Weiter bei Phase 3.
Falls noch 0en, aber keine 1en: Verwerfe
Falls noch 1en, aber keine 0en: Verwerfe

Falls weder 1en noch 0en: akzeptiere
Phase 3: Laufe einmal über das Band, lösche die erste 0 und die erste 1.
Weiter bei Phase 2.

M_1 entscheidet L. Wieviele Schritte braucht M auf Eingabe der Länge n?

Phase 1: Anzahl der Schritte: $n+n$, also $O(n)$
Phase 2/3: pro Durchlauf: $\underbrace{2(n+n)}_{\text{Kopfpositionen}}$, also $O(n)$
Anzahl der Durchläufe: $\frac{n}{2}$
Anzahl Schritte insgesamt: $\frac{n}{2}O(n) = (O(n^2))$
Die Schrittanzahl von M_1 auf Eingabe der Länge n ist also $O(n^2)$

Bemerkung: Die O-Notation erlaubt es, gewisse Details aus der Maschinenbeschreibung bzw. -analyse zu vernachlässigen.
(Vergröberung der Sicht, Zugeständnis an die Durchführbarkeit der Analyse.)

Definition 4.1.2 *Sei* $f : \mathbb{N} \to \mathbb{N}$. *Die Zeitkomplexitätsklasse* $\texttt{TIME}(t(n))$ *ist definiert als* $\texttt{TIME}(t(n)) = \{L \mid L$ *ist Sprache, welche von einer* $O(t(n))$*-zeitbeschränkten Turingmaschine entschieden wird*$\}$.

Zurück zur oben diskutierten Sprache L: Wir wissen: $L \in \texttt{TIME}(n^2)$
Frage: Gibt es eine Turingmaschine, die L schneller entscheidet?
M_1 kann verbessert werden, etwa indem in Phase 3 jeweils zwei 1en und zwei 0en gelöscht werden, aber dies beeinflußt nicht die asymptotische Laufzeit.

Betrachte M_2, welche wie folgt arbeitet:

Phase 1: Wie bei M_1.
Phase 2: Wie bei M_1.
Phase 3: Prüfe, ob Anzahl 0en und 1en zusammen gerade ist.
Falls nein, verwerfe, sonst weiter bei Phase 4.
Phase 4: Lösche jede zweite 0 und jede zweite 1.

M_2 entscheidet L (Begründung: zu Beginn von Phase 2 gilt stets: Eingabe ist in L gdw. Anzahl der 0en und 1en auf dem Band ist gleich).

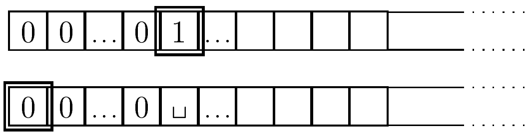
Laufzeit: Schrittanzahl auf Eingabe der Länge n:

Phase 1: $O(n)$
Phase 2,3,4: $(\log n)O(n) = O(n \log n)$

(Anzahl der Durchläufe multipliziert mit dem Zeitaufwand für einen Durchlauf)
Also: M_2 ist $O(n \log n)$-zeitbeschränkte Turingmaschine
Diese Laufzeitschranke ist optimal (für 1-Band-Turingmaschinen)[ohne Beweis].

Frage: Welche Rolle spielt das zugrundegelegte Maschinenmodell (hier 1-Band Turingmaschine)?

Wir zeigen, dass sich beim Übergang zu 2 Bändern noch eine Verbesserung ergibt: 2-Band-Turingmaschine M_3 arbeite wie folgt:



Phase 1: Wie bei M_1.
Phase 2: Laufe mit Kopf 1 bis zur ersten 1.
Kopiere dabei 0en auf Band 2.
Repositioniere Kopf 2 an den Anfang des Bandes.
Phase 3: Laufe mit Kopf 1 und 2 simultan bis zum ersten ␣.
Falls beide Köpfe das erste ␣ zugleich erreichen, so akzeptiere, sonst verwerfe.

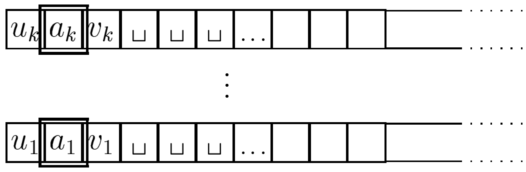
M_3 ist eine $O(n)$-zeitbeschränkte 2-Band-Turingmaschine, die L entscheidet.

Beobachtung: Ein Wechsel des zugrunde gelegten Maschinenmodells bedeutet eine Veränderung der Zeitkomplexität von Sprachen / Problemen.

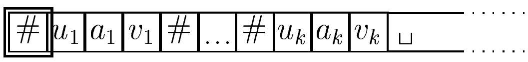
Lösung: Weitere Vergröberung der Analyse, damit das zugrunde gelegte Maschinenmodell irrelevant wird.

Satz 4.1.1 *Sei $t : \mathbb{N} \to \mathbb{N}$ mit $t(n) \geq n$ für alle n.*
Zu jeder $t(n)$-Zeit k-Band-Turingmaschine gibt es eine äquivalente $O(t^2(n))$-zeitbeschränkte 1-Band-Turingmaschine.

Beweisskizze: Analyse des Beweises von Satz 1.4.1.
Sei M eine $t(n)$-Zeit-k-Band-Turingmaschine.
Typische Konfiguration:



Konstruiere M' wie im Beweis von Satz 1.4.1.
Typische M'-Konfiguration:



Für die Simulation eines M-Schrittes muss M' den beschriebenen Teil des Arbeitsbandes etwa 4 mal durchlaufen (Erfassen aller unterstrichenen Positionen und ggfs. Verrücken eines Teils der Inschrift nach rechts).
Die Inschrift des Arbeitsbandes von M' beim Lauf einer Eingabe der Länge n hat höchstens die Länge $k(t(n)+1) \subseteq O(t(n))$.
Die Simulation eines M-Schrittes kostet höchstens $O(t(n))$ M'-Schritte.
Schrittzahl insgesamt: $O(t(n)) \cdot O(t(n)) = O(t^2(n))$.

□

4.2 Die Klasse P

Wir identifizieren hier „Probleme“ mit Sprachen (d.h. mit Entscheidungsproblemen) und fassen diejenigen zusammen, die durch eine $p(n)$-zeitbeschränkte TM (für ein Polynom p) entschieden werden können:

$$P = \bigcup_{p \text{ Polynom}} \texttt{TIME}(p(n))$$

These von Edmonds, Copham ($\sim$ 1965)

Eine Sprache L ist im praktischen Sinne („effizient“) entscheidbar genau dann, wenn $L \in P$.

Argumente dafür:

1. Der Bezug auf Turingmaschinen ist unwesentlich. Andere Standardformalismen (Programme) sind durch TM in Polynomzeit simulierbar. Genauer: Für Eingabegröße n werde ein Programmschritt durch $q(n)$ TM-Schritte (mit Polynom q) simuliert. Aus einem $p(n)$-zeitbeschränkten Programm wird dann eine $p(q(n))$-zeitbeschränkte TM (polynomielle Laufzeit).

2. Zwischen polynomiellem und exponentiellem Zeitaufwand liegt ein Qualitätssprung.

Illustration:

Wachstum	Eingabegröße n			
	10	30	60	100
n	0,01 ms			0,1 ms
n^2	0,1 ms	0,9 ms	3,6 ms	10 ms
n^3	1 ms	27 ms	0,2s	1 s
2^n	1 s	1,7 min	36000 Jahre	$4 \cdot 10^{15}$ Jahre

Einwände:

1. Die (Hardware-)Beschleunigung mildert das Bild ab. Dies trifft jedoch nicht zu. Bei fester Rechenzeit (z.B. 1 Tag), erlaubt Beschleunigung um Faktor 1000 beim Wachstum

n: Behandlung von 1000 mal so großen Eingaben möglich,
n^2: Behandlung von 30 mal so großen Eingaben möglich,
n^3: Behandlung von 10 mal so großen Eingaben möglich,
2^n: Vergrößerung der Eingabegröße um 10 möglich.

2. Die mittleren Laufzeiten (statt der schlechtesten) für gegebene Eingabegröße können auch für $O(2^n)$-Algorithmen praktikabel sein, oder die Konstante c in der Abschätzung $c \cdot 2^n (= O(2^n))$ ist sehr klein. Für manche Probleme ist dies ein wesentlicher Gesichtspunkt. Die These gilt daher nicht so absolut wie die Church-Turing-These.

Weitere Beispielprobleme

1. **Graph-Erreichbarkeitsproblem**

   Gegeben: Endlicher, gerichteter Graph $G = (V, E)$, Knoten $s, t \in V$.
   Frage: Gibt es einen Pfad durch G von s nach t?

   Standardlösung
   Führe Breiten- oder Tiefensuche von s aus durch und markiere die dabei besuchten Knoten, teste, ob irgendwann t markiert wird. Zeitaufwand $O(|V| + |E|)$ (Polynomzeitalgorithmus).

   Umsetzung auf TM'en

   a) **Kodierung des Problems durch Sprache**, d.h. von G, s, t jeweils durch Wort $w_{G,s,t}$. Darstellung von $G = (\{1, \dots, n\}, E)$ durch Wortlisten:

   $$w_G := \mathrm{bin}(1), \dots, \mathrm{bin}(n)|\mathrm{Kanten}(\underbrace{\mathrm{bin}(1), \mathrm{bin}(j)}_{(1,j)\in E})|\mathrm{Kanten}(\underbrace{\mathrm{bin}(2), \mathrm{bin}(j)}_{(2,j)\in E})| \dots |\mathrm{Kanten}(\mathrm{bin}(n), \mathrm{bin}(j))$$

   $$w_{G,s,t} = w_G||\mathrm{bin}(s), \mathrm{bin}(t)$$

   Länge von $w_{G,s,t} = O(\underbrace{n}_{|V|} \underbrace{\log(n)}_{\text{1 Knoten}} + \underbrace{n^2}_{|E|} \underbrace{2\log(n)}_{\text{eine Kante}} + 2\log(n)) = O(n^3)$.

   Entsprechende Sprache: $\{w_{G,s,t} \mid \text{es existiert Pfad durch } G \text{ von } s \text{ nach } t\}$.

   b) **Durchführung der Breitensuche durch TM** ($s = 1, t = 2$)

      i. Markiere bin(1) überall in w_G (z.B. Ersatz von 0,1 durch $\underline{0}$,$\underline{1}$).
      ii. Solange im vorherigen Durchgang Markierung erfolgte, mache weiteren Markierungsdurchgang: Gehe Kantenliste durch, markiere bin(j) überall, falls (bin(i),bin(j)) auftaucht mit bin(i) markiert, bin(j) unmarkiert.
      iii. Teste, ob bin(2)($= t$) markiert ist.

   Aufwand:

      i. Vergleich von bin(i) mit bin(1) (am Anfang) jeweils Feststellung, ob bin(i) = bin(1). Buchstabenweiser Vergleich mit erstem Eintrag bin(1).
      $O(\underbrace{n^2 \log(n)}_{|W_G|} \cdot \underbrace{\log(n)}_{|bin(i)|})$
      ii. $O(nn^2n^2 \log(n) \log(n))$

iii. $O(n^2 \log(n))$

Insgesamt erhalten wir eine $O(n^7)$-zeitbeschränkte Turingmaschine. Ab jetzt geschieht Zeitaufwandsanalyse auf intuitiver Ebene, statt mit Bezug auf TM'en.

2. **Knoten-Erzeugbarkeitsproblem**

   Gegeben: Graph $G = (V, E), S \subseteq V, t \in V$.

   Die Menge der aus S erzeugbaren Knoten ist induktiv definiert durch:

   (a) Jeder Knoten in S ist aus S erzeugbar.

   (b) Sind $(v_1, v), \ldots, (v_k, v)$ die Kanten nach v und sind $v_1, \ldots, v_k$ aus S erzeugbar, so auch v.

   Frage: Ist t erzeugbar ?

   Dieses Problem tritt an vielen Stellen der Informatik auf, in unterschiedlichen Verkleidungen – überall dort, wo man feststellen will, ob ein Element t zu einer induktiv definierten Menge gehört. Man löst es durch einen *Markierungsalgorithmus*:
   Markiere alle Elemente von S, dann alle diejenigen, die gemäß (b) aus schon markierten Knoten gewonnen werden können, und dies wiederholt, bis sich keine neuen Markierungen mehr ergeben. Hat der betrachtete Graph n Knoten, terminiert das Verfahren nach $\leq n$ Markierungsdurchgängen, und bei jedem Durchgang muß man jeden von $\leq n$ möglichen Zielknoten auf die Erzeugbarkeit aus ($\leq n$ vielen) schon markierten Quellknoten überprüfen. Damit benötigt der Markierungsalgorithmus einen Zeitaufwand von $O(\underbrace{n}_{\text{Durchgänge}} \cdot \underbrace{n}_{\text{Zielknoten}} \cdot \underbrace{n}_{\text{Quellknoten}})$ Schritten, ist also polynomial.

Übungen

Übung 36 Ein Graph (V, E) heißt *ungerichtet*, falls E irreflexiv und symmetrisch ist.
Sei $k \geq 1$. Eine *k-Clique* eines ungerichteten Graphen $G = (V, E)$ ist eine k-elementige Teilmenge $U \subseteq V$ mit $(u, v) \in E$ für alle paarweise verschiedenen $u, v \in U$.
Wir betrachten das folgende Problem $\underline{k\text{-Clique}}$:

Gegeben: Ungerichteter Graph $G = (V, E)$.

Frage: Besitzt G eine k-Clique?

Zeigen Sie:

(a) $\underline{\text{3-Clique}}$ ist in P.

(b) $\underline{k\text{-Clique}}$ ist in P für jedes gegebene $k \geq 1$.

Geben Sie jeweils zunächst eine knappe, umgangssprachliche Beschreibung der Arbeitsweise der jeweiligen Maschine und danach eine detailliertere Beschreibung mit einigen Beispielkonfigurationen an.

Übung 37 Sei UNARY-PRIMES $= \{1^p \mid p \text{ ist Primzahl}\}$.
Zeigen Sie: $\underline{\text{UNARY-PRIMES}}$ ist in P.
Skizzieren Sie dazu die Arbeitsweise einer entsprechenden TM.

4.3 Die Klasse NP

Wir beginnen mit einem Beispiel zur Motivation, dem Problem der k-Färbbarkeit von Graphen.

Definition 4.3.1 *Sei $G = (V, E)$ ungerichteter Graph, also E irreflexiv und symmetrisch. Eine k-Färbung von G ist eine Partition $V_1, \ldots, V_n$ von V, so dass*
für $(u, v) \in E$ jeweils u, v zu verschiedenen Mengen aus $V_1, \ldots, V_k$ gehören. $(*)$

Problem 3-Färbbarkeit
Gegeben: Ungerichteter Graph G
Frage: Hat G eine 3-Färbung?

Lösungsansatz: durch Ausprobieren aller Partitionen (V_1, V_2, V_3) mit jeweiliger Überprüfung, ob $(*)$ erfüllt ist.

Aufwand: $\underbrace{\text{Anzahl der Partitionen } f : V \to \{1,2,3\}}_{3^{|V|}} \cdot \underbrace{\text{Aufwand zur Überprüfung auf } (*)}_{(+)}$

Zu (+):
Gegeben: Graph $G = (V, E)$, Partition V_1, V_2, V_3 von V.
Frage: Ist (V_1, V_2, V_3) eine 3-Färbung, das heißt: gilt $(*)$?
Test durch Überprüfung jeder Kante, d.h. der Zeitaufwand $\leq c \cdot |V| \cdot |V|$ ist polynomial in $|V|$.
Insgesamt ergibt sich exponentieller Aufwand.

Wir betrachten nun die Ebene der Turingmaschine, also die
formale Version von (+) auf Ebene der Wörter über $\Sigma_{GCode} = \{0, 1, (,), ,, /\}$.
Gegeben: Wortpaar $(u, v) \in \Sigma^*_{GCode} \times \Sigma^*_{GCode}$.
Frage: Ist u der Code w_G eines Graphen G (gemäß Darstellung der Knoten mit Kantenliste) und v Code einer Partition aus 3 Klassen (Binärdarstellung der Knoten durch **,** getrennt und durch / in drei Abschnitte unterteilt) so, dass die Partition $(*)$ erfüllt?

Die Paare (u, v), die die Antwort „ja“ verlangen, bilden eine Relation $R_{3F} \subseteq \Sigma^*_{GCode} \times \Sigma^*_{GCode}$. Relation R_{3F} ist durch Turingmaschine in Polynomzeit entscheidbar.

Definition 4.3.2 *Eine Relation $R \subseteq \Sigma^*_1 \times \ldots \times \Sigma^*_n$ heißt polynomial entscheidbar, falls TM M existiert, die R entscheidet, d.h.*

$$M : q_0 w_1 \sqcup w_2 \ldots \sqcup w_n \to \begin{cases} \text{Ausgabe 1} & \text{, falls } (w_1, \ldots, w_n) \in R \\ \text{Ausgabe 0} & \text{, falls } (w_1, \ldots, w_n) \notin R \end{cases}$$

und ferner Polynom p, so dass M angesetzt auf $w_1 \sqcup \ldots \sqcup w_n$ nach $\leq p(|w_1| + \ldots + |w_n|)$ Schritten stoppt.

Beispiel: R_{3F} ist polynomial entscheidbar.

Bemerkung: G hat 3-Färbung $\Leftrightarrow \exists v(|v| \leq |w_G| \land (w_G, v) \in R_{3F})$.
Diese Formulierung zeigt eine typische Gestalt von Problemen:
Es ist zur Eingabe w eine „Lösung“ v zu finden, deren Korrektheit in Polynomzeit getestet werden kann (R_{3F} ist polynomial entscheidbar). Doch sind die Lösungskandidaten v nur

durch die Bedingung $|v| \leq |w|$ eingegrenzt; damit wächst die *Anzahl* dieser v exponentiell in $n := |w|$ (Größe der Eingabe), und die „Suche durch Ausprobieren“ ist folglich nicht praktikabel. Wir fassen derartige Probleme zur Komplexitätsklasse NP zusammen (wobei wir $|v|$ durch eine polynomiale Schranke in $|w|$ statt durch $|w|$ beschränken):

Definition 4.3.3 (Klasse NP) *Eine Sprache $L \subseteq \Sigma^*$ gehört zur Klasse NP, falls es für geeignetes Alphabet Σ' eine polynomial entscheidbare Relation $R \subseteq \Sigma^* \times \Sigma'^*$ und ein Polynom $q(n)$ gibt mit*

$$w \in L \Leftrightarrow \exists v(|v| \leq q(|w|) \wedge (w, v) \in R).$$

Ein allgemeines Entscheidungsproblem gehört zu NP (bzw. P), wenn eine natürliche Kodierung als Sprache zu NP (bzw. P) gehört.

Beispiel: Das 3-Färbbarkeitsproblem wird kodiert durch
$L_{3F} := \{w_G | w_G$ kodiert ungerichteten Graphen, der 3-Färbung besitzt$\}$, d.h. wir haben

$$w_G \in L_{3F} \;\Leftrightarrow\; \exists v(|v| \leq |w_G| \;\wedge\; (w_G, v) \in R_{3F}).$$

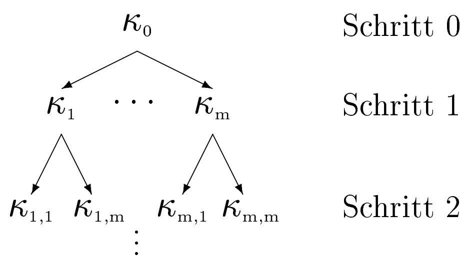
Da $R_{3F} \in P$, gilt $L_{3F} \in$ NP.

Die folgenden Überlegungen dienen der Rechtfertigung des Namens NP: *Nichtdeterministische Turingmaschinen* (NTM's) können ebenfalls zur Definition von NP herangezogen werden.
Bisheriges Turingmaschinenmodell: deterministisch (DTM). NP steht für „nichtdeterministisch in Polynomzeit entscheidbar“. Hier die genauen Definitionen:

Definition 4.3.4 *Eine* **nichtdeterministische Turingmaschine** *(NTM) hat die Form $M = (Q, \Sigma, \Gamma, \delta, q_s, q_s)$ wie zuvor bei TM'en, jedoch mit $\delta : Q \times \Gamma \to Pot(Q \times \Gamma \times \{L,R,N\})$*

Schreibweise: $\delta(q, a) = \{(q_1, a_1, d_1), \ldots, (q_m, a_m, d_m)\}$ (Verzweigungsgrad m).

Entsprechend hat eine Konfiguration gegebenenfalls mehrere Folgekonfigurationen. Es ergibt sich bei gegebener Startkonfiguration κ_0 der folgende Konfigurationsbaum:



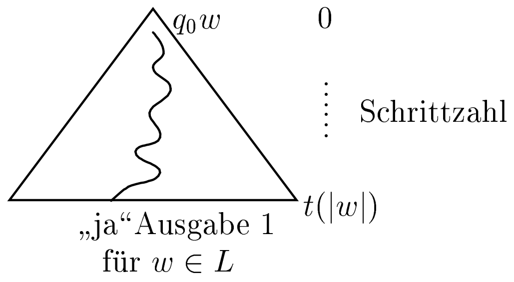
Eine *TM-Berechnung* ist wie zuvor eine Konfigurationsfolge $\kappa_0, \kappa_1, \ldots, \kappa_l$ mit: κ_{i+1} ist eine Folgekonfiguration von κ_i $(i < l)$.

Definition 4.3.5 Eine NTM M entscheidet $L \subseteq \Sigma^*$, *wenn für Eingabe* $w \in \Sigma^*$ *jeweils jede Berechnung von* $\kappa_0 = q_0 w$ *aus terminiert und*

$$w \in L \Leftrightarrow \text{wenigstens eine } \textit{Berechnung von } q_0 w \textit{ aus liefert Ausgabe 1.}$$

Eine NTM M heißt $t(n)$-zeitbeschränkt, *wenn M angesetzt auf $w \in \Sigma^*$ mit jeder möglichen Berechnung nach $\leq t(|w|)$ Schritten stoppt.*

Bild des Konfigurationsbaums einer $t(n)$-zeitbeschränkten NTM, die L entscheidet:



Definition 4.3.6 $\texttt{NTIME}(t(n)) = \{L | L$ *wird durch $t(n)$-zeitbeschränkte NTM entschieden*$\}$.

Wir untersuchen die $t(n)$-zeitbeschränkten NTM's in zwei Etappen:

1. Vergleich mit DTM's.
2. Angekündigte Charakterisierung der Klasse NP durch DTM's:
   $\text{NP} = \bigcup_{p \text{ Polynom}} \texttt{NTIME}(p(n))$.

Zunächst zum Vergleich NTM/DTM:

Satz 4.3.1 (über NTM's/DTM's ohne Zeitkomplexität) *Wird L durch eine NTM entschieden, so auch durch eine DTM. Kurz: „Zu jeder NTM M gibt es eine äquivalente DTM M“. (NTM, DTM sind „äquivalent“, falls sie dieselbe Sprache entscheiden).*

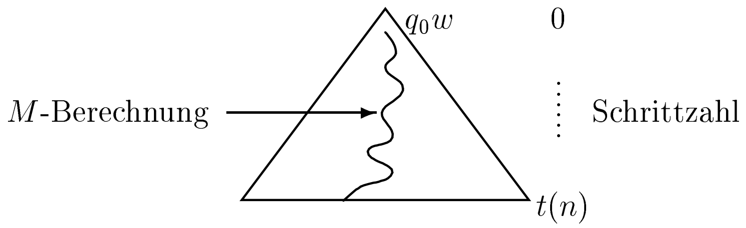
Beweisidee: Sei M NTM mit maximalem Verzweigungsgrad m. Zur Eingabe w (Anfangskonfiguration $q_0 w$) wird eine Berechnung eindeutig festgelegt durch eine Indexfolge $i_0 i_1 \dots i_l$ (l Schritte weit) mit $i_j \in \{1, \dots, m\}$, wobei i_j=Index des ausgewählten Tripels nach j Schritten.

□

Ansatz für gesuchte DTM:

Gesuchte DTM M' hat 3 Bänder: für Eingabe, für die genannten Indexfolgen (sukzessiv erzeugt in kanonischer Reihenfolge), für Simulation von M.
M' terminiert, wenn für geeignetes l zu keiner Indexfolge $i_0 i_1 \dots i_l$ eine der NTM M entsprechende Berechnung durchführen kann und zwar mit Ausgabe „ja“ gdw. M zuvor einmal „ja“ ausgegeben hat, sonst „nein“.

Wir geben genauere Darstellung für den Fall, dass M $t(n)$-zeitbeschränkt ist. In diesem Fall genügt die Herstellung aller Indexfolgen der Länge $t(n) - 1$. Zu jeweiliger Indexfolge $i_0 \dots i_{t(n)-1}$ führe Simulation so weit durch, wie gemäß der Indizes möglich.



□

Aufwand bei Eingabe w:

(a) jeweils Herstellung einer neuen Indexfolge: $O(t(|w|))$

(b) Simulation von M zu jeweiliger Indexfolge: $O(t(|w|))$
Anzahl der Indexfolgen der Länge $t(n)$: $m^{t(n)}$

Insgesamt: $O(t(|w|) \cdot m^{t(|w|)})$
Reduktion auf ein Band liefert Aufwand $O((t(|w|) \cdot m^{t(|w|)})^2)$.
Hieraus ergibt sich:

Satz 4.3.2 *Eine $t(n)$-zeitbeschränkte NTM ist durch eine $2^{O(t(n))}$-zeitbeschränkte DTM simulierbar.*
(Genauer gilt dies nur für sogenannte „zeitkonstruierbare Funktionen“ t, wie z.B. Polynome)

Nun zeigen wir:

$$\bigcup_{p \text{ Polynom}} \texttt{NTIME}(p(n)) \quad \stackrel{!}{=} \quad \text{NP}$$

Satz 4.3.3 *Eine Sprache $L \subseteq \Sigma^*$ wird durch eine $p(n)$-zeitbeschränkte NTM entschieden (wobei p Polynom) $\Leftrightarrow$ L hat Darstellung $L = \{w \in \Sigma^* \mid \exists v(|v| \leq q(|w|) \wedge (w,v) \in R)\}$ für geeignetes Polynom q und eine polynomial entscheidbare Relation $R \subseteq \Sigma^* \times \Sigma'^*$*

Beweis: „$\Rightarrow$“
Sei $M = (Q, \Sigma, \Gamma, \delta, q_0, q_s)$ NTM, $p(n)$ zeitbeschränkt, die L entscheidet. Wir identifizieren eine akzeptierende M-Berechnung zur Eingabe w (also eine Konfiguration $\underbrace{\kappa_0}_{q_0w}, \kappa_1, \ldots, \kappa_l$) mit dem Wort $\beta(w) = \kappa_0 \# \kappa_1 \# \ldots \# \kappa_l \#$ über $\Gamma \cup Q \cup \{\#\}$.

□

Es gelten folgende Bedingungen:

(1) $l \leq p(|w|)$ (denn M ist $p(n)$-zeitbeschränkt)

(2) $|\kappa_0| = |w| + 1$

(3) $|\beta(w)| \leq (|w| + 1 + p(|w|)) \cdot p(|w|) + \underbrace{p(|w|)}_{\text{Anzahl } \#} \leq q(|w|)$ für ein Polynom q.

Die Aussage (3) ergibt sich so:
Nach Arbeitsweise von Turingmaschinen ist $|\kappa_{i+1}| \leq |\kappa_i| + 1$ für $i < l$, also hat jede Konfiguration κ_i höchstens die Länge $|w| + 1 + p(|w|)$ (maximale in $p(|w|)$ Schritten erreichbare Länge). Diese Länge wird mit der *Anzahl* $p(|w|)$ der κ_i multipliziert, und es wird der Längenbeitrag der Zeichen # addiert.
Die Gesamtlänge lässt sich also durch ein Polynom $q(|w|)$ abschätzen.
Die Relation $R_M \subseteq \Sigma^* \times (\Gamma \cup Q \cup \{\#\})^*$ werde definiert durch

$$(w, v) \in R_M :\Leftrightarrow v \text{ ist akzeptierende } M\text{-Berechnung zu } w.$$

Dann ist klar:

$$w \in L \Leftrightarrow \exists v(|v| \leq q(|w|) \wedge (w, v) \in R_M).$$

Es bleibt zu prüfen, dass R_M durch eine polynomial zeitbeschränkte DTM entschieden wird.

Arbeitsweise von DTM, die R_M entscheidet, bei Eingabe $w_\sqcup v$:

1. Prüfe, ob v mit $q_0 w\#$ beginnt; falls ja, gehe auf # (als Arbeitsfeld).

2. Solange möglich wiederhole: Vergleiche maximales #-freies Wort vor dem Arbeitsfeld mit dem maximalen #-freien Wort nach Arbeitsfeld und prüfe, ob das zweite Wort Folgekonfiguration vom ersten ist.

3. Falls # letztes Symbol ist: Überprüfe, ob Q-Buchstabe davor q_s ist und ob anschließend eine 1 steht. In diesem Fall gib „ja“ aus, sonst „nein“.

Der Zeitaufwand für diesen Fall ist polynomial, nämlich quadratisch in der Eingabelänge.

Zur Beweisrichtung „$\Leftarrow$“ :
$R \subseteq \Sigma^* \times \Sigma'^*$ werde entschieden durch $p(n)$-zeitbeschränkte DTM M_R, wobei p Polynom sei. Es gelte $(*)$: $w \in L \;\Leftrightarrow\; \exists v(|v| \leq q(|w|) \;\wedge\; (w, v) \in R)$. O.B.d.A. hat gegebenes Polynom q die Form $c \cdot n^k$ (mit c, k geeignet groß).

Arbeitsweise der gesuchten NTM M, die L entscheidet (bei Eingabe w):

1. stelle aus w das Wort $w_\sqcup|^{c\cdot|w|^k}$ her (Die Striche markieren genug Platz für alle Kandidaten v)

2. „Rate“ ein v der Länge $\leq c\cdot|w|^k$, d.h. überschreibe das Wort $|^{c|w|^k}$ nichtdeterministisch durch Σ'-Buchstaben, ggfs. mit $\sqcup$ ab gewisser Stelle

3. Auf $w_\sqcup v$ (hergestellt) arbeite wie M_R und übernehme die Antwort.

Diese NTM M entscheidet wegen $(*)$ die Sprache L; sie ist außerdem polynomial zeitbeschränkt.

Zum Zeitaufwand:

zu 1. (a) Stelle aus w das Wort $w_\sqcup|^{|w|}$ her.

(b) Wiederhole $(k-1)$-mal:

Stelle aus $w_\sqcup|^m$ jeweils das Wort $w_\sqcup|^{m\cdot|w|}$ her,
füge also $(|w|-1)$-mal den Block $|^m$ an.

Aufwand: $O(m \cdot |w| \cdot (|w|-1))$, $m \leq |w|^{k-1}$, also polynomial in $|w|$.

(c) Analog: Übergang von $|^{|w|^k}$ zu $|^{c|w|^k}$, auch polynomial.

Zusammen ergibt sich also polynomialer Aufwand.

zu 2. Benutze folgende Transitionen, die die Strichfolge in Σ'−Buchstaben mit nachfolgenden $\sqcup$'s umwandeln: $\delta_M(q, |) = \{(q, b_1, R), \dots, (q, b_l, R), (q', \sqcup, R)\}$.
$\delta_M(q', |) = (q', \sqcup, R)$ (Hierbei $\Sigma' = \{b_1, \dots, b_l\}$).
Aufwand: $O(q(|w|))$.

zu 3. Aufwand für M_R auf Input $w_\sqcup w$: $r(|w| + q(|w|))$, also polynomial.

Insgesamt ist also der Zeitaufwand aller Phasen zusammen polynomial in $|w|$.

□

Übungen

Übung 38 Definition: Sei $k \geq 1$. Eine NTM M habe *Verzweigungsgrad* k falls es für alle Konfigurationen von M höchstens k Folgekonfigurationen gibt.

Zeigen Sie: Zu jeder polynomzeitbeschränktem NTM gibt es eine äquivalente polynomzeitbeschränkte NTM mit Verzweigungsgrad 2.

Übung 39 Erinnerung: Für zwei Sprachen $L_1, L_2 \subseteq \Sigma^*$ definiert man

$$L_1 \cdot L_2 = \{uv \mid u \in L_1, v \in L_2\}.$$

Zeigen Sie:

(a) Sind L_1, L_2 in P, so auch $L_1 \cdot L_2$.

(b) Sind L_1, L_2 in NP, so auch $L_1 \cdot L_2$.

Geben Sie jeweils zunächst eine knappe, umgangsprachliche Beschreibung der Arbeitsweise der jeweiligen Maschine und danach eine detailliertere Beschreibung mit einigen Beispielkonfigurationen an.

Übung 40 Sei $L \subseteq \Sigma^*$ eine Sprache, p ein Polynom und M eine L akzeptierende NTM mit: für alle $w \in L$ gibt es *eine* M-Berechnung der Länge $\leq p(|w|)$.

Zeigen Sie: L in NP.

4.4 NP-vollständige Probleme

Es ist bis heute offen, ob die Probleme in NP (Suchprobleme mit exponentiell großem Suchraum) tatsächlich in Polynomzeit entscheidbar sind, also bereits zu P gehören.
Die Frage

$$P \quad = \quad \text{NP ?}$$

ist vielleicht das wichtigste offene Problem in der theoretischen Informatik.
Immerhin kann man konkrete Probleme L in NP angeben, die für diese Frage „typisch“ sind, für die also gilt

$$L \in P \quad \Leftrightarrow \quad P \;=\; \text{NP}$$

Die „typischen“ Probleme in NP sind „innerhalb von NP von maximaler Schwierigkeit“. Um das zu präzisieren, führen wir einen passenden Reduktionsbegriff ein, analog zu den berechenbaren Reduktionen im vorangehenden Kapitel:

Definition 4.4.1 (Polynomzeit-Reduzierbarkeit) *Seien Σ_1, Σ_2 Alphabete, $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$. L_1 heißt* polynomzeitreduzierbar *auf L_2 ($L_1 \leq_p L_2$), falls eine polynomzeitberechenbare Funktion $f : \Sigma_1^* \to \Sigma_2^*$ existiert mit: $w \in L_1 \Leftrightarrow f(w) \in L_2$ für alle $w \in \Sigma_1^*$.*

Bemerkung (vgl. Übung 41):

(a) $L_1 \leq_p L_2$, $L_2 \in \mathrm{P} \Rightarrow L_1 \in \mathrm{P}$.

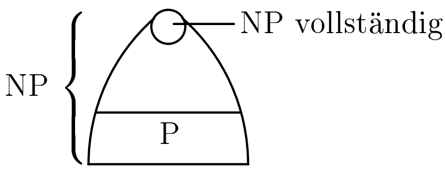
(b) $L_1 \leq_p L_2$, $L_2 \in \mathrm{NP} \Rightarrow L_1 \in \mathrm{NP}$.

Definition 4.4.2 *L_0 heißt NP-vollständig, falls*

(1) $L_0 \in NP$.

(2) $\forall L \in NP : L \leq_p L_0$.

Eine NP-vollständige Sprache ist also in NP und „mindestens so schwierig“ wie jede andere Sprache in NP.
Zur Illustration diene das folgende Bild:



Wir wissen aber nicht, ob – wie angedeutet – tatsächlich $\mathrm{P} \subsetneq \mathrm{NP}$ gilt.

Bemerkung: Sei L_0 NP-vollständig. Dann gilt: $\mathrm{P} \subsetneq \mathrm{NP} \Leftrightarrow L_0 \in \mathrm{NP} \backslash \mathrm{P}$

Beweis:

„$\Leftarrow$“ : trivial

„$\Rightarrow$“ : $L_0 \in$ NP ist klar nach Voraussetzung über L_0.

Wäre $L_0 \in P$, so gälte für alle $L \in$ NP: $L \leq_p L_0 \in$ P. Nach vorausgehender Bemerkung, Teil (a), ist dann $L \in$ P. Widerspruch!

□

Wir haben mit NP-vollständigen Sprachen argumentiert, ohne die Existenzfrage zu klären: Gibt es überhaupt NP-vollständige Sprachen?

Wir werden zeigen:

Satz 4.4.1 (von Cook, Levin (1971)) *Das Erfüllbarkeitsproblem der Aussagenlogik*

Gegeben:	*Aussagenlogische Formel φ*
Frage:	*Ist φ erfüllbar?*

ist NP-vollständig.

Wir führen die entsprechenden Begriffe nun genauer ein.

Definition 4.4.3 Aussagenlogische Formeln *sind induktiv definiert durch*

- *Jede Variable xi (i natürliche Zahl in Binärdarstellung) ist eine aussagenlogische Formel.*
- *Sind φ_1, φ_2 aussagenlogische Formeln, so auch $\neg\varphi_1, (\varphi_1 \vee \varphi_2), (\varphi_1 \wedge \varphi_2)$.*

Notation:

- Statt $\neg xi$ auch $\overline{xi}$
- Klammer-Ersparung mit Konventionen: $\neg$ bindet stärker als $\wedge$, $\wedge$ stärker als $\vee$, Außenklammern dürfen wegfallen.

Beispiele: $(x10 \wedge \neg x10) \vee x11$, $x0 \wedge x1 \wedge (\neg x11 \vee \neg x0)$ sind aussagenlogische Formeln.

Definition 4.4.4 *Eine aussagenlogische* **Formel in KNF (konjunktiver Normalform)** *hat die Form $c_1 \wedge \ldots \wedge c_m$ mit $c_i = l_{i_1} \vee \ldots \vee l_{i_n}$, wobei l_{i_j} eine Variable x_k oder eine negierte Variable $\neg x_k$ ist. Man nennt die l_{i_j} auch* Literale*, die c_i* Klauseln*.*

Definition 4.4.5 *Eine* Belegung *für φ ordnet jeder in φ vorkommenden Variablen einen der Werte 0,1 zu. Eine Belegung* erfüllt *φ, falls die wie üblich definierte Auswertung von φ unter dieser Belegung den Wert 1 ergibt.*

Beispiel: $\varphi = \underbrace{(\underbrace{x10}_{1} \wedge \underbrace{\neg \underbrace{x10}_{1}}_{0})}_{0} \vee \underbrace{x11}_{0}$ mit Belegung $x10 \mapsto 1, x11 \mapsto 0$ liefert den Wert 0.

Diese Belegung erfüllt φ also nicht.

Definition 4.4.6 *φ heißt* erfüllbar*, falls es eine φ erfüllende Belegung gibt.*

Beispiel: $\varphi = (x10 \wedge \neg x10) \vee x11$ ist erfüllbar (mit Belegung $x10 \mapsto 1, x11 \mapsto 1$).

Wir kodieren nun das Erfüllbarkeitsproblem durch eine Sprache über dem Alphabet

$$\Sigma_{\text{BoolF}} = \{x, 0, 1, (,), \neg, \vee, \wedge\}.$$

Definition 4.4.7 SAT $:= \{\varphi \in \Sigma^*_{BoolF} \mid \varphi$ *erfüllbare aussagenlogische Formel in KNF*$\}$.

Damit erhalten wir folgende Neuformulierung des Satzes von Cook-Levin.

Satz 4.4.2 (von Cook, Levin (1971)) *<u>SAT</u> ist NP-vollständig.*

Beweis: In zwei „Etappen“:

1. <u>SAT</u> $\in$ NP.
2. $\forall L \in \text{NP} : L \leq_p$ <u>SAT</u>.

Punkt 1 ist einfach (vgl. Übung 43).
Zu 2.: Sei $L \in$ NP. Wähle NTM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_s)$, Polynom p so, dass M $p(n)$-zeitbeschränkt ist und M die Sprache L entscheidet. Dann haben wir:

$w \in L \Leftrightarrow$ $(*)$ es existiert eine M-Berechnung aus $\leq p(|w|)$ Konfigurationen, die mit $q_0 w$ beginnt und mit Ausgabe 1 terminiert.

Gesucht: Polynomzeitbeschränkte Funktion $f : \Sigma^* \to \Sigma^*_{\text{BoolF}}, w \mapsto f(w) := \varphi_w$ mit:
$w \in L \Leftrightarrow \varphi_w \in$ SAT, d.h. $w \in L \Leftrightarrow (+)$ φ_w erfüllbar.

Analyse von $(*)$:
Wir setzen $Q = \{q_0, \ldots, q_m\}$, $q_s = q_m$, $\Gamma = \{\underbrace{a_1, \ldots, a_k}_{\Sigma}, \ldots, a_l\}$, $C := Q \mathbin{\dot\cup} \Gamma \cup \{\#\}$.
M sei so modifiziert, dass die Konfiguration uq_s1v immer wieder reproduziert wird.

$(+)$ bedeutet, dass ein Konfigurationsschema folgender Form existiert ($w = b_1 \ldots b_n$):

Schrittzahl \ Bandfelder	0	1	2	...	$p(n)+3$	$p(n)+4$
0	#	q_0	b_1	$\ldots b_n \sqcup \cdots$	$\sqcup$	#
$\vdots$	...			$\vdots$		
$\vdots$	#			$q_s 1$		#
$p(n)$	#			$q_s 1$		#

Wir nennen es „M-Tableau zur Eingabe w“.

Ansatz für φ_w: φ_w soll ausdrücken, dass es ein solches Konfigurationsschema gibt (wir sprechen kurz von einem akzeptierenden M-Tableau zur Eingabe w). Benutze Variablen x_{ijb}, deren Gültigkeit ausdrücken soll: Zelle (i, j) ist beschriftet mit b.

Die Formel q_w besteht aus vier größeren Teilen:
Hierbei sei $\varphi_w := \varphi_{\text{Zellen}} \wedge \varphi_{\text{Start}} \wedge \varphi_{\text{Schritt}} \wedge \varphi_{\text{Akzeptiert}}$

$$\varphi_{\text{Zellen}} = \bigwedge_{\substack{0\le i\le p(n)\\ 0\le j\le p(n)+4}} \left(\bigvee_{b\in C} x_{ijb} \wedge \bigwedge_{\substack{b,b'\in C\\ b\ne b'}} \neg(x_{ijb} \wedge x_{ijb'}) \right).$$

Erfüllbare Belegung für φ_{Zellen} fixiert eine Beschriftung des Tableaus, mit genau einem $b \in C$ auf jeder der Positionen (i,j).
Wenn man die Konjunktions- und Disjunktionszeichen umgangssprachlich als Quantoren formuliert, sagt die Formel: „Für jede Position (i,j) gibt es ein b welches dort steht, und für je zwei Buchstaben $b' \neq b''$ steht nicht sowohl b als durch b' auf (i,j)“.
φ_{Start} beschreibt die erste Zeile des Schemas (für $w = b_1 \dots b_n$):

$$\varphi_{\text{Start}} = x_{00\#} \wedge x_{01q_0} \wedge x_{02b_1} \wedge x_{03b_2} \wedge \dots \wedge x_{0n+1b_n} \wedge x_{0n+2\sqcup} \wedge \dots x_{0p(n)+3\sqcup} \wedge x_{0p(n)+4\#}$$

$\varphi_{\text{Akzeptiert}}$ behauptet die Existenz einer Position (i,j), die mit dem Stoppzustand beschriftet ist, so dass das (Ausgabe-)Symbol 1 rechts daneben steht (M stoppt mit Ausgabe „ja“):

$$\varphi_{\text{Akzeptiert}} = \bigvee_{\substack{0\le i\le p(n)\\ 0\le j< p(n)+4}} (x_{ijq_s} \wedge x_{ij+1\,1}).$$

$\varphi_{\text{Akzeptiert}}$ beschreibt die korrekte Fortsetzung der Beschriftung von Zeile zu Zeile.

Es genügt die Festlegung zulässiger 2×3-Fenster in dem Schema:

a_1	a_2	a_3
a_4	a_5	a_6

Hier ein Beispiel für den Fall, dass M den Befehl $qaq'a'R$ ausführen kann.

Zulässige Fenster sind z.B.

b	q	a
b	a'	q'

c	b	q
c	b	a'

c	c	b
c	c	b

a	b	a
q'	b	a

Nicht zulässige Fenster sind z.B.

b	b	a
b	a	a

c	q	a
c	b	q'

c	q	a
q	c	c

Ein 2×3-Fenster heißt zulässig, wenn es gemäß Turingtafel von M als Segment eines M-Tableaus auftreten kann.

Nun kann man folgendes zeigen:
Bemerkung: Enthält eine Buchstabenmatrix der Größe $(p(n)+1) \times (p(n)+5)$ in der 1. Zeile eine M-Startkonfiguration zur Eingabe w, und ist jedes 2×3-Fenster der Matrix zulässig, dann ist die Matrix ein M-Tableau zur Eingabe w.
(Beweis durch Induktion über die Zeilennummern.) Also setzen wir

$$\varphi_{\text{Schritt}} := \bigwedge_{\substack{0\le i\le p(n)\\ 0< j< p(n)+4}} \left(\bigvee_{\substack{a_1\cdots a_6\\ \text{zuläss. Fenster}}} \left(x_{ij-1a_1} \wedge x_{ija_2} \wedge x_{ij+1a_3} \wedge x_{i+1j-1a_4} \wedge x_{i+1ja_5} \wedge x_{i+1j+1a_6} \right) \right).$$

Die Indizierung der 6 Felder des (2×3)-Fensters folgt dem folgenden Muster:

$ij-1$	ij	$ij+1$
$i+1j-1$	$i+1j$	$i+1j+1$

Es gilt:

$$w \in L \Leftrightarrow M \text{ akzeptiert } w \Leftrightarrow \varphi_w \text{ erfüllbar} \Leftrightarrow \varphi_w \in \text{SAT}.$$

Also ist unsere Transformation $w \mapsto \varphi_w$ eine korrekte Reduktion.

Für die Behauptung $L \leq_p$ SAT fehlt noch, dass $w \mapsto \varphi_w$ polynomzeitberechenbar ist. Wir prüfen, dass die *Länge* von (φ_w) polynomial beschränkt in $|w|$ ist. (Etwas mehr Fleißarbeit zeigt dann, dass eine Turingmaschine tatsächlich in Polynomzeit diese Ausgabe φ_w herstellen kann.)

1. Länge einer Variablen als Wort:
   Wir benötigen so viele Variablen wie es Tripel (i, j, b) gibt mit $0 \leq i \leq p(n)$, $0 \leq j \leq p(n) + 4$, $b \in C$, insgesamt also $N := (p(n) + 1) \cdot (p(n) + 5) \cdot |C|$ viele. Hierzu werden Binärzahlen der Länge $\log N$ benötigt; zusammen mit dem Buchstaben x hat eine Variable also die Länge $l := 1 + \log N$.
   l ist polynomial in n.

2. $$\begin{array}{lcl} |\varphi_{\text{Zellen}}| & \leq & (p(n)+1)\cdot(p(n)+5)\cdot(2|c|\cdot l + |c|^2\cdot 6l) \\ |\varphi_{\text{Start}}| & \leq & (p(n)+5)\cdot l \\ |\varphi_{\text{Akzeptiert}}| & \leq & (p(n)+1)\cdot(p(n)+5)\cdot 5l \\ |\varphi_{\text{Schritt}}| & \leq & (p(n)+1)\cdot(p(n)+5)\cdot |c|^6\cdot 14l \end{array}$$
   Insgesamt ist die Summe beschränkt durch ein Polynom $q(w)$.

Mit SAT haben wir nun ein erstes NP-vollständiges Problem in der Hand. Weitere NP-vollständige Probleme gewinnen wir durch Reduktionen, völlig analog zum Nachweis der Unentscheidbarkeit von Problemen aus schon bekannten unentscheidbaren Problemen.
Wir verwenden hier die Polynomzeit-Reduzierbarkeit $\leq_p$ (an Stelle von $\leq$ im vorangehenden Kapitel). Zwei Probleme behandeln wir als Beispiele:
Eine Spezialisierung von SAT (das Problem 3SAT) und das Problem der 3-Färbbarkeit von Graphen, das wir als Einstieg unserer Untersuchungen zu NP betrachtet haben.

Problem 3SAT: Gegeben: Aussagenlogischer Ausdruck φ in 3KNF (d.h. in konjunktiver Normalform mit ≤ 3 Literalen pro Klausel), d.h.
$\varphi = c_1 \wedge \ldots \wedge c_m$, mit $c_i = l_{i1} \vee l_{i2} \vee l_{i3}$, wobei l_{ij} von der Form x_k oder $\neg x_k$ ist.
Frage: Ist φ erfüllbar?

Satz 4.4.3 *3SAT ist NP-vollständig.*

Beweis: Zwei Behauptungen sind zu zeigen:

1. 3SAT $\in$ NP. (Dies ist klar, da bereits das allgemeinere Problem SAT zu NP gehört.)

2. $\forall L \in \text{NP} : L \leq_p$ 3SAT.

Ansatz: Wir modifizieren die Formel φ_w aus der Transformation $w \mapsto \varphi_w$ des vorangehenden Satzes zu einer Formel ψ_w in 3KNF.

1. Etappe: Herstellung der KNF (noch ohne Berücksichtigung der Literal-Anzahl pro Klausel) Da φ_w bereits eine Konjunktion von $\varphi_{\text{Zellen}}, \varphi_{\text{Schritt}}, \varphi_{\text{Start}}, \varphi_{\text{Akzeptieren}}$ ist, können wir die Bestandteile getrennt betrachten.

- φ_{Zellen}: Es liegt hier eine KNF vor, bis auf die Subformeln $\neg(x_{ijb} \wedge x_{ijb'})$. Diese aber sind äquivalent zu den Klauseln $(\neg x_{ijb} \vee \neg x_{ijb'})$.
- φ_{Schritt} : Hier müssen wir die Formeln $\bigvee_{a_1 \dots a_6}(x_{i\,j-1\,a_1} \wedge \dots \wedge x_{i+1\,j+1\,a_6})$ als Konjunktionen von Disjunktionen schreiben. Dies ist möglich mit dem Distributivgesetz $(a \wedge b) \vee (c \wedge d) \equiv (a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$.
- $\varphi_{\text{Akzeptiert}}$: Analog.
- φ_{Start} : Liegt schon in KNF vor.

Beachte: Hierdurch ergibt sich nur eine polynomiale Vergrößerung der Länge.

2. Etappe: Transformation der KNF-Formel in erfüllbarkeits-äquivalente 3KNF-Formel.
Gegeben: Klauseln $c_1 \wedge \dots \wedge c_m$ mit c_i der Form $l_1 \vee \dots \vee l_r$. Betrachte den einzig interessanten Fall $r > 3$:
Stelle aus $l_1 \vee \dots \vee l_r$ eine Konjunktion von Klauseln mit nur 3 Literalen, mit neuen Hilfsvariablen $z_1 \dots z_{r-2}$, her.
Statt $l_1 \vee \dots \vee l_r$ schreibe:

$$(l_1 \vee l_2 \vee z_1) \wedge (\neg z_1 \vee l_3 \vee z_2) \wedge (\neg z_2 \vee l_4 \vee z_3) \wedge \dots \wedge (\neg z_{r-2} \vee l_{r-1} \vee l_r).$$

Dann ergibt sich sofort: $c_1 \wedge \dots \wedge c_m$ erfüllbar gdw. die transformierte Formel ist erfüllbar.
□

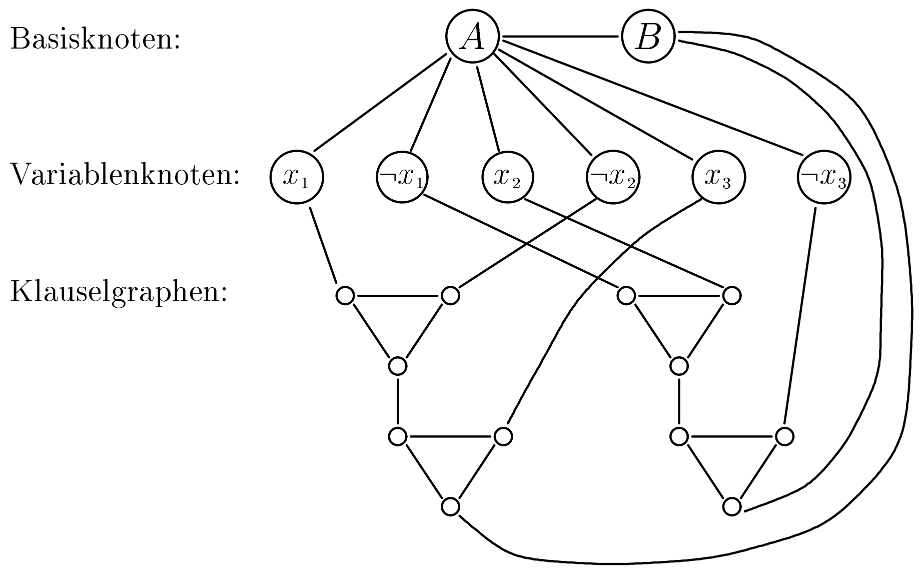
Satz 4.4.4 *Das Problem $\underline{\text{COLOR(3)}}$ der 3-Färbbarkeit von Graphen ist NP-vollständig.*

Beweis: Zugehörigkeit von $\underline{\text{COLOR(3)}}$ zu NP haben wir im vorangehenden Abschnitt schon gezeigt. Es genügt: $\underline{\text{3SAT}} \leq_p \underline{\text{COLOR(3)}}$.
Wir finden eine polynomzeitberechenbare Transformation, die eine Formel φ in 3KNF in einen Graphen G_φ so überführt, dass

$$\varphi \text{ erfüllbar } \Leftrightarrow G_\varphi \text{ ist 3-färbbar.}$$

Betrachte $\varphi = c_1 \wedge \dots \wedge c_m$, wobei $c_i = l_{i1} \vee l_{i2} \vee l_{i3}$, l_{ij} jeweils x_k oder $\neg x_k$.
Beispiel: $\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$
Wir geben G_φ erst einmal für dieses Beispiel an.

G_φ hat folgende Struktur:



Bei Variablen $x_1, \ldots, x_n$ gibt es entsprechend $2n$ Variablenknoten; hat φ m Klauseln, so gibt es entsprechend m Klauselgraphen. Die Anbindung der oberen Knoten eines Klauselgraphen an die Variablenknoten erfolgt einfach durch Kanten zu den Knoten x_i bzw. $\neg x_i$, die in der betrachteten Klausel vorkommen. Für die Klausel $c = l_1 \vee l_2 \vee l_3$ sprechen wir dann von der Anbindung bei den Knoten $l_1,\ l_2,\ l_3$.

Zeige: φ erfüllbar $\Leftrightarrow$ G_φ 3-färbbar

Beweis: „$\Rightarrow$“

Sei $\beta : \{x_1, \ldots, x_n\} \to \{0, 1\}$ eine φ erfüllende Belegung.

Finde 3-Färbung von G_φ. Nehme Farben 0,1,2.

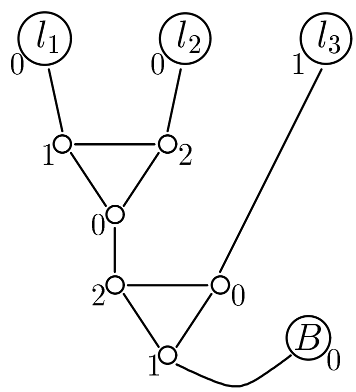
Färbe A mit 2, B mit 0.

Färbe Knoten x_i mit Farbe 1, falls $\beta(x_i) = 1$, sonst mit 0, $\neg x_i$ dann mit 0 bzw. 1.

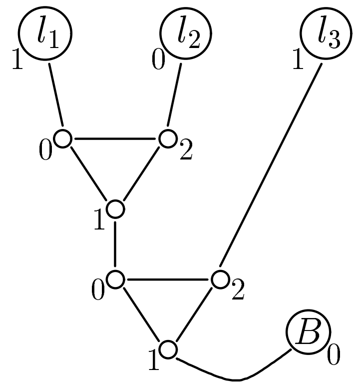
Da β die Formel φ erfüllt, wird für jeden Klauselgraphen, mit Anbindung etwa bei l_1, l_2, l_3, einer der Knoten l_1, l_2, l_3 mit 1 gefärbt.

1. Fall:

Die Knoten $l_1,\ l_2$ der Anbindung haben beide die Farbe 0. Dann ist l_3 mit 1 gefärbt, und wir können den Klauselgraphen färben:

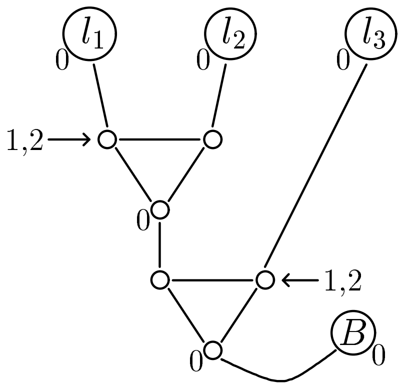


2. Fall:
Bei l_1, l_2 kommt wenigstens eine 1 vor. Dann können wir die oberen Knoten des oberen Dreiecks mit 0 und 2 färben (die Farbe 0 bei demjenigen Knoten, der an den mit 1 gefärbten Literalknoten angebunden ist).
Wiederum ist die Färbung auf den ganzen Klauselgraphen fortsetzbar. (Beachte, dass l_3 mit 0 oder mit 1 gefärbt sein darf.)



„⇐"
Sei G_φ 3-färbbar. Finde erfüllende Belegung von φ.
Nehme Farben 0,1,2. Ohne Beschränkung der Allgemeinheit sei 2 auf A, 0 auf B. Dann sind die Variablenknoten mit 0,1 gefärbt, x_i, $\neg x_i$ jeweils mit verschiedenen Farben.
Definiere die Belegung β durch $\beta(x_i) :=$ Farbe von x_i.

Zeige: Dieses β erfüllt φ. (Dann ist φ erfüllbar.)
Es genügt: In jeder Klausel $l_1 \vee l_2 \vee l_3$ bekommt wenigstens ein Literal den Wert 1. D.h.: Im Klauselgraphen, der zur Klausel $l_1 \vee l_2 \vee l_3$ gehört, führt wenigstens eine Kante zu einem mit 1 gefärbten Variablenknoten l_i.
Annahme: Alle drei Variablenknoten haben Farbe 0.
Dann ergeben sich auf den oberen Knoten des oberen Dreiecks notwendigerweise die Farben 1,2, darunter also Farbe 0, auf den oberen Knoten des unteren Dreiecks wieder die Farben 1,2, und somit darunter wieder die Farbe 0. Da Knoten B mit 0 gefärbt war, ist dies ein Widerspruch zur Voraussetzung, dass eine Färbung vorliegt.



□

Die Probleme SAT, 3SAT und COLOR(3) sind nur erste Beispiele für NP-vollständige Probleme – man kennt inzwischen tausende davon.
Allen ist gemeinsam, dass man sie bisher nur mit der Methode der erschöpfenden Suche in einem exponentiell großen Suchraum lösen kann, ohne eine Richtschnur zu haben, die diese Suche „zielgerichtet abkürzt“. Ferner würde ein Polynomzeit-Algorithmus für nur ein einziges NP-vollständiges Problem L_0 genügen, um *alle* Probleme aus NP in Polynomzeit lösen zu können. (Denn für ein $L \in$ NP gilt dann ja $L \leq_p L_0$, und wäre $L_0 \in P$, so auch L.)
Viele glauben, dass es nicht möglich ist, die NP-vollständigen Probleme in Polynomzeit zu lösen, aber einen Beweis dafür hat man eben nicht.
Häufig betrachtet man erweiterte Fragestellungen; dann handelt es sich um Optimierungsprobleme an Stelle von Entscheidungsproblemen. Hierbei geht es um die Berechnung einer Kostengröße statt eines Wertes 0,1 („nein“/„ja“).

Beispiel: Gegeben: Graph G (ungerichtet)
Finde kleinstes k, für das G k-färbbar ist.

Dies läßt sich auf eine Kette von Entscheidungsproblemen zurückführen, d.h. in unserem Beispiel auf die Frage der k-Färbbarkeit für $k = 2, 3, \ldots$.
Die Schwierigkeit der NP-vollständigen Probleme befreit die Informatik nicht davon, dennoch nach praktisch durchführbaren Lösungsansätzen zu suchen. Dies ist Gegenstand eigener Forschungsgebiete (und Vorlesungen).

Wir nennen einige solche Ansätze:

1. Heuristiken (Auswahl des lokalen Optimums)
2. Approximationsalgorithmen
   Idee: Finde in Polynomzeit Optimum bis auf konstanten Fehler.
3. Probabilistische Algorithmen (Randomisierte Algorithmen)
   Idee: Würfele bei Laufzeit des polynomzeitbeschränkten Algorithmus und garantiere Korrektheit der Antwort mit Mindestwahrscheinlichkeit.
4. Molekularbiologie
   Idee: Repräsentiere die Lösungskandidaten durch DNA-Moleküle und finde das Optimum durch biologische/chemische/physikalische Prozesse.
   (Adleman hat ein Traveling-Salesman-Problem mit dieser Methode gelöst.)

Übungen

Übung 41 Seien $L_0, L_1, L_2 \subseteq \Sigma^*$ Sprachen. Zeigen Sie:

(a) Falls $L_2 \in$ P und $L_1 \leq_p L_2$, so $L_1 \in$ P.

(b) Falls $L_2 \in$ NP und $L_1 \leq_p L_2$, so $L_1 \in$ NP.

(c) Falls P = NP und $L_0 \in$ P aber $L_0 \notin \{\emptyset, \Sigma^*\}$, so ist L_0 NP-vollständig.

Übung 42 Ein ungerichteter Graph $G = (V, E)$ heiße *Euler'sch*, falls es einen geschlossenen Pfad durch G gibt, auf dem jede Kante genau einmal vorkommt; mit anderen Worten falls es ein n gibt und $v_0, \dots, v_n \in V$ mit $v_0 = v_n$ und für alle $e \in E$ existiert genau ein $i \in \{0, \dots, n-1\}$ mit $e \in \{(v_i, v_{i+1}), (v_{i+1}, v_i)\}$.
Sei EULER die Menge aller Kodierungen von Euler'schen Graphen.
Zeigen Sie *unter Rückgriff auf die Definition von NP*, dass EULER in NP ist. Geben Sie also ein Polynom q und eine polynomzeit-berechenbare Relation R an mit:

$$w \in L \iff \exists v(|v| \leq q(|w|) \wedge (w, v) \in R).$$

Übung 43 Zeigen Sie, dass das Problem KNF-SAT in NP ist:

Gegeben: Aussagenlogischer Ausdruck α in KNF (konjunktiver Normalform).

Frage: Ist α erfüllbar?

Übung 44 Zeigen Sie, dass KNF-SAT polynomzeit-reduzierbar ist auf das 10. Hilbert'sche Problem.
Hinweis: Geben Sie eine polynomzeit-berechenbare Funktion f derart an, dass für einen KNF-Ausdruck $\beta(x_1, \dots, x_n)$ gilt: $f(\beta(x_1, \dots, x_n))$ ist ein Polynom in den Variablen $x_1, \dots, x_n$ mit Koeffizienten aus $\mathbb{Z}$ derart, dass $\beta(x_1, \dots, x_n)$ erfüllbar ist gdw. $f(\beta(x_1, \dots, x_n))$ hat eine Nullstelle in $\mathbb{Z}$.
Statt der Polynomzeit-Berechenbarkeit von f reicht es, wenn sie die (schwächere) Aussage, dass jedes f-Bild polynomielle Größe in der Länge des Arguments hat, begründen.

Übung 45 Sei CLIQUE folgendes Problem:

Gegeben: Graph G und $k \geq 1$ in Binärdarstellung.

Frage: Hat G eine Clique der Größe k?

Zeigen Sie: CLIQUE ist NP-vollständig.
Hinweis: Finden Sie ein Polynomzeitreduktion von 3KNF-SAT auf CLIQUE.
Überlegen Sie sich den genauen Unterschied zu Übung 36!

Übung 46 Sei DNF-SAT folgendes Problem:

Gegeben: Aussagenlogische Formel α in disjunktiver Normalform.

Frage: Ist α erfüllbar?

Zeigen Sie: DNF-SAT $\in$ P.

Übung 47 Definition: Eine „Hornklausel" ist eine aussagenlogische Formel der Form

$$\neg y_1 \vee \ldots \vee \neg y_n \vee z \qquad (n \geq 0)$$

oder der Form

$$\neg y_1 \vee \ldots \vee \neg y_n \qquad (n \geq 1),$$

wobei jeweils $y_1, \dots, y_n$ aussagenlogische Variablen sind.

Sei HSAT folgendes Problem:

Gegeben: Konjunktion α von Hornklauseln.

Frage: Ist α erfüllbar?

Zeigen Sie, dass HSAT in P ist.
Hinweise: (1) $\neg y_1 \vee \ldots \vee \neg y_n \vee z$ ist logisch äquivalent zu $(y_1 \wedge \ldots \wedge y_n) \rightarrow z$.
(2) Finden Sie eine Polynomzeitreduktion von HSAT auf das Problem der „Knotenerzeugbarkeit" (Abschnitt 4.2).

Übung 48 Geben Sie eine Folge $(\alpha_n)_{n\geq 1}$ von aussagenlogischen Ausdrücken in KNF an so, dass die Länge von α_n polynomial in n ist und dass es keine Folge von äquivalenten Formeln in DNF gibt, deren Länge polynomial in n ist.
Bemerkung: Die Existenz von solchen Ausdrücken ist der Grund dafür, dass aus Übung 46 („DNF-SAT $\in P$") und Übung 43 („KNF-SAT ist NP-vollständig") nicht P = NP folgt.

Kapitel 5

Ausblick und Rückblick

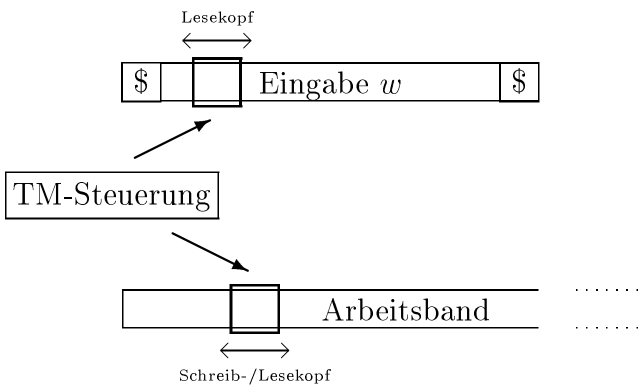
In einigen abschließenden Bemerkungen skizzieren wir

- die Grundlagen zur „Platzkomplexität“ (bei der der Speicherplatzverbrauch an Stelle des Zeitaufwands untersucht wird),
- einige Aspekte zur Einordnung der Berechenbarkeits- und Komplexitätstheorie in den allgemeinen Rahmen der Informatik.

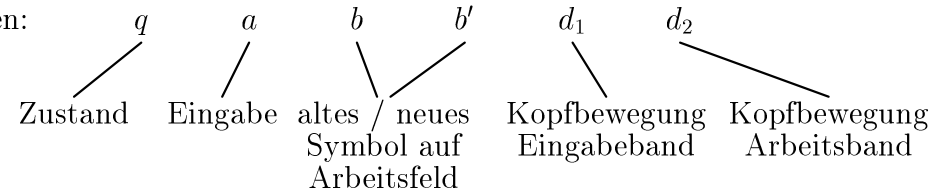
5.1 Platzkomplexität

Hier trennt man im Turingmaschinen-Modell den „Platz für die Eingabe“ vom „Platz für die Arbeit“.

Definition 5.1.1 *Eine* **Offline-Turingmaschine** *(deterministisch oder nichtdeterministisch) hat ein Leseband für die Eingabe und ein Arbeitsband:*



Form der Befehlszeilen:



M heißt $f(n)$-*platzbeschränkt*, falls für eine Eingabe der Länge n höchstens $O(f(n))$ viele Felder des Arbeitsbandes besucht werden.
Es sei DSPACE$(f(n))$ = Klasse der Sprachen L, die durch $f(n)$-platzbeschränkte deterministische Turingmaschinen entschieden werden, analog NSPACE$(f(n))$ bei Bezug auf nichtdeterministische Turingmaschinen.

Prominente Beispiele von Platzkomplexitätsklassen:

$$\begin{array}{lcl} \text{DLOG} & = & \text{DSPACE}(\log(n)) \\ \text{NLOG} & = & \text{NSPACE}(\log(n)) \\ \text{PSPACE} & = & \bigcup\limits_{p \text{ Polynom}} \text{DSPACE}(p(n)) \end{array}$$

(Dies ist, wie man zeigen kann, gleich der Klasse $\bigcup\limits_{p \text{ Polynom}} \text{NSPACE}(p(n))$.)

Beispiel für ein Problem in NLOG: Graph-Erreichbarkeitsproblem
Gegeben: Kodierung eines Graphen G und zweier ausgezeichneter Knoten s, t.
Frage: Gibt es einen Pfad durch G von s nach t?.
Als Problemgröße vereinbaren wir die Anzahl n der Knoten.
(Kodierung des Graphen durch Wort w_G hat größere Länge; logarithmischer Platz in n liefert sofort logarithmischen Platz in $|w_G|$ (Länge der Kodierung von G).)

Satz 5.1.1 *Das Erreichbarkeitsproblem für Graphen gehört zu NLOG.*

Beweis: Zur Eingabe Code w_G von Graph G (G habe n Knoten, Knotennamen aus je $\log(n)$ Bits) und Knoten s, t arbeite die gesuchte NTM (offline) wie folgt:
Die Arbeitsbandinschrift enthält jeweils drei Daten: Zählerstand z (von 1 bis n), zwei Knotennamen (Platzverbrauch also $3 \log n$).
Initialisierung: $z = 1, n_1 = s, v_1 = v$ (v_1 nichtdeterministisch geraten).
Prüfe, ob (s, v) Kante von G ist. Wenn nicht, stoppe mit Ausgabe „nein“. Falls ja, prüfe, ob $v = t$ ist. Wenn ja, stoppe mit Ausgabe „ja“.
Übergang von z, u, v jeweils zu $z + 1, v, v'$ (v' nichtdeterministisch geraten). Prüfe, ob (v, v') Kante, und wenn dies gilt, ob $v' = t$. Abbruch mit Ausgabe „nein“ bei Zählerüberlauf (Länge $> \log(n)$, also Pfadlänge $> n$). Dann existiert eine TM-Berechnung mit stop bei Ausgabe „ja“ genau dann, wenn in G ein Pfad von s nach t existiert.

$\square$

In einer zweiten Etappe diskutieren wir ein für die Klasse PSPACE typisches Problem. Es betrifft die sog. „quantifizierten Booleschen Formeln“.

Definition 5.1.2 *Quantifizierte Boolesche Formeln entstehen aus aussagenlogischen Ausdrücken $\varphi(x_1, \ldots, x_n)$ durch Voranstellen von Quantoren $\exists x_i$ und $\forall x_i$.*

$\exists x \varphi(x)$ besagt: „Es existiert Wahrheitswert für x so, dass hiermit φ wahr wird“. Die Festlegung der Bedeutung im allgemein Fall erfolgt durch folgende Äquivalenz:

$$\exists x_1 \varphi(x_1, x_2 \ldots, x_n) \text{ sei äquivalent zu } \varphi(0, x_2, \ldots, x_n) \vee \varphi(1, x_2, \ldots, x_n)$$

$$\forall x_1 \varphi(x_1, x_2 \ldots, x_n) \text{ sei äquivalent zu } \varphi(0, x_2, \ldots, x_n) \wedge \varphi(1, x_2, \ldots, x_n)$$

Bemerkung: Sind alle Variablen quantifiziert, heißt die Formel *voll quantifiziert*, und die Formel ist dann wahr oder falsch (ohne Vorgabe einer Belegung).

Beispiel:

$\forall x_1 \exists x_2 ((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2))$ wahr (wähle zu Wert für x_1 jeweils Wert für $x_2 = \neg$(Wert für x_1).

$\exists x_1 \forall x_2 ((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2))$ falsch

Das Problem SAT läßt sich als Sonderfall hiervon formulieren: Eine Formel $\varphi(x_1, \ldots, x_n)$ (ohne Boolesche Quantoren) ist erfüllbar $\Leftrightarrow \exists x_1 \ldots \exists x_n \varphi(x_1, \ldots, x_n)$ wahr.

Problem QBF:

Gegeben:	Voll quantifizierte Boolesche Formel φ.
Frage:	Ist φ wahr ?

Satz 5.1.2 *Das Auswertungsproblem für quantifizierte Boolesche Formeln ist in PSPACE.*

Beweisidee: QBF gehört zu PSPACE. Betrachte z.B. $\exists x_1 \forall x_2 \exists x_3 \ldots \exists x_n \underbrace{\varphi_0(x_1, \ldots, x_n)}_{\text{quantorenfrei}}$.

Rekursive Prozedur arbeitet sukzessive die Quantoren ab.

Beispiel: Bei Abarbeitung von $\exists x_i$ rufe Prozedur zweimal auf, für $x_i = 0$ und für $x_i = 1$, bilde „$\vee$“ der Ergebnisse.
Bei Abarbeitung von $\forall x_i$ rufe Prozedur wieder zweimal auf mit $x_i = 0, x_i = 1$ und bilde „$\wedge$“ der Ergebnisse.
Als Speicher genügt ein Stack von n Bits (bei Bearbeitung von $\exists / \forall x_i$ sind jeweils die Bits für $x_1, \ldots, x_{i-1}$ nötig). Wir erhalten linearen Platzbedarf in Anzahl n der Variablen, erst recht in der Länge der Formel.

Natürlich ist der *Zeitaufwand* exponentiell, da die Aufrufe alle Wahrscheinlichkeitswertkombinationen für $x_1, \ldots x_n$ erfassen.

□

Definition 5.1.3 *Eine Sprache L_0 heißt PSPACE-vollständig, falls*

(a) $L_0 \in$ *PSPACE.*

(b) $\forall L \in$ *PSPACE:* $L \leq_p L_0$.

Satz 5.1.3 *QBF ist PSPACE-vollständig.*

Der Beweis verläuft ähnlich wie der Nachweis der NP-Vollständigkeit von SAT.

Vergleich Zeit-Platz-Komplexität

Bemerkungen zu DSPACE$(f(n)) \subseteq$ DTIME$2^{O(f(n))}$, NSPACE$(f(n)) \subseteq$ NTIME$2^{O(f(n))}$
Zum Beweisansatz betrachte D/N TM mit Arbeitsbandschranke $f(n)$: Die Anzahl der möglichen Konfigurationen (für $\mathfrak{A} = (Q, \Sigma, \Gamma, q_0, q_s, \Delta)$) ist

$$|\Gamma|^{f(n)} \cdot f(n) \cdot |Q| \cdot n$$

(Die Faktoren erfassen die Maximalzahlen der möglichen Arbeitsbandinschriften, Kopfpositionen auf Arbeitsband, Zustände.)

Da D/N TM für jede Eingabe gemäß Definition mit jeder Berechnung stoppt, erfolgt die Termination für geeignete Konstanten c, d in $\leq cnf(n)d^{f(n)}$ Schritten, also ist die behauptete Zeitschranke eingehalten.

Folgerung: NLOG $\subseteq$ P $\subseteq$ NP $\subseteq$ PSPACE
Die Echtheit aller Inklusionen ist offen, außer dass NLOG $\subset$ PSPACE bekannt ist. Auch weiß man: Es gelten Hierarchiesätze für Platz und Zeit, z.B.

$$\mathrm{NSPACE}(\log n) \subsetneq \mathrm{NSPACE}(n) \subsetneq \mathrm{NSPACE}(n^2) \subsetneq \ldots$$

Übungen

Übung 49 Zeigen Sie: <u>QBF</u> in PSPACE. (Führen Sie die Beweisskizze von Satz 5.1.2.)

Übung 50 Zeigen Sie: $\{wcw \in \{0,1,c\}^* \mid w \in \{0,1\}^*\} \in$ DLOG.

Übung 51 Eine *TM mit Ausgabeband* ist eine Version einer TM, die 3 separate Bänder hat:

- ein Eingabeband, auf dem die Eingabe in der üblichen Weise präsentiert wird und auf dem nicht geschrieben werden darf,
- ein Arbeitsband, auf dem anfangs Blanks stehen,
- ein Ausgabeband, auf dem nur gedruckt wird und dessen Kopf sich nach dem Drucken eines Symbols um ein Feld nach rechts (und sonst gar nicht) bewegt (ebenfalls initial mit $\sqcup$'s beschrieben).

Eine TM mit Ausgabeband *berechnet* eine Funktion $f : \Sigma^* \to \Gamma^*$, falls für alle $w \in \Sigma^*$ gilt, dass M, gestartet mit Eingabe w, terminiert gdw. $f(w)$ ist definiert, und in diesem Falle $f(w)$ die Inschrift des Ausgabebandes beim Erreichen einer Stoppkonfiguration ist.
Platzkomplexität für solche TM mit Ausgabeband ist wie üblich mit Bezug allein auf das Arbeitsband definiert. Eine totale Funktion $f : \Sigma^* \to \Gamma^*$ ist *DLOG-berechenbar* wenn es eine $\mathcal{O}(\log n)$-platzbeschränkte TM mit Ausgabeband gibt, die f berechnet.
Hiermit definieren wir folgenden Reduktionsbegriff: Seien $L_1 \subseteq \Sigma^*$ und $L_2 \subseteq \Gamma^*$ Sprachen. Es gelte $L_1 \leq_{\log} L_2$ gdw. es gibt ein DLOG-berechenbare Funktion $f : \Sigma^* \to \Gamma^*$ derart, dass für alle $w \in \Sigma^*$ gilt: $w \in L_1 \iff f(w) \in L_2$.
Zeigen Sie: Wenn $L_1 \leq_{\log} L_2$ und $L_2 \in$ P, so folgt $L_1 \in$ P.

5.2 Was haben wir erreicht?

Was haben wir in dieser Vorlesung erreicht? Hoffentlich ein tieferes Verständnis für die informatischen Grundbegriffe „algorithmisches Problem“ und „algorithmische Lösung“: Algorithmische Probleme haben wir durch (Wort-)Funktionen f und (im Falle von Entscheidungsproblemen) durch Wortmengen, also formale Sprachen L erfaßt.

Eine Wortfunktion repräsentiert ein gewünschtes Eingabe-Ausgabe-Verhalten, und eine Sprache kennzeichnet die Instanzen eines Entscheidungsproblems, die die Antwort „ja“ verlangen. Wir haben die fundamentalen Fragen

- Gibt es eine Lösung?
- Gibt es eine effiziente Lösung?

studiert, d.h. die Fragen nach der Berechenbarkeit von Funktionen und der Entscheidbarkeit von Sprachen, bzw. nach der Komplexität von Problemen. Präzise Antworten wurden ermöglicht durch den Bezug auf präzise Formalismen (Turingmaschinen und Mini-Programmiersprachen), und die Adäquatheit dieser Formalismen haben wir begründet. Insbesondere haben wir interessante Beispiele für Probleme gefunden, die algorithmisch *nicht* lösbar sind, z.B. das Dominoproblem.

Eine weitgehende Aussage konnten wir mit dem Satz von Rice (3.2.3) treffen:

Alle interessanten Entscheidungsprobleme über Programme (Algorithmen) sind unentscheidbar.

Bei der Unterscheidung zwischen effizient lösbaren und nur nicht-effizient lösbaren Problemen haben wir zwar mit der Klasse P der in Polynomzeit lösbaren Probleme eine vernünftige Präzisierung gewonnen, doch ist für eine weitere Problemklasse (die Probleme in NP) der Status (d.h. die Zugehörigkeit / Nicht-Zugehörigkeit zu P) noch ungeklärt. Immerhin haben wir mit den NP-vollständigen Problemen gute Kandidaten für „schwierige Probleme“ gewonnen.

Trotz dieser Einsichten erfaßt diese Theorie, insbesondere was die Definition von „Problem“ angeht, nur einen – wenn auch fundamentalen – Ausschnitt der Problemstellungen der Informatik insgesamt. Häufig ist eine allgemeinere Sicht vonnöten. Auf zwei Arten von erweiterten Theorien sei hingewiesen:

(1) In vielen Anwendungen sind die betrachteten Daten nicht Wörter, sondern komplexe Strukturen, wie Netzwerke, graphische Darstellungen usw. In diesen Situationen, etwa in Fragen zur Datenbanktheorie oder zur Bildverarbeitung, wird durch die Kodierung der Daten mittels Wörtern die Problemstellung „verbogen“ – man möchte die Fragen der Berechenbarkeit und Komplexität eigentlich direkter studieren. Immerhin gibt es eine solche Berechenbarkeits- und Komplexitätstheorie über Relationalstrukturen im Sinne der mathematischen Logik (Strukturen, wie man sie z.B. in der Theorie der Datenbanken braucht); diese Theorie wird in der sog. „deskriptiven Komplexitätstheorie“ entwickelt.

(2) Die in dieser Vorlesung betrachteten Berechnungsprobleme werden durch Funktionen f : Eingabebereich $E \to$ Ausgabebereich A beschrieben, und Algorithmen sollen entsprechend für eine (passende) Eingabe $e \in E$ mit einer Ausgabe $a \in A$ terminieren. Viele Informatik-Systeme (Betriebssysteme, Kommunikationsprotokolle etc.) stehen jedoch in *fortwährender* Wechselwirkung mit ihrer Umgebung und sollen in der Regel

nicht terminieren. Ein entsprechendes „Berechnungsproblem für reaktive Systeme" ist also durch eine Menge *unendlicher Systemläufe*, d.h. Folgen von abwechselnden Eingaben und Ausgaben beschrieben. Man betrachtet also eine Menge R von Folgen $e_0 a_0 e_1 a_1 \ldots$ mit $e_i \in E$, $a_i \in A$, in der die „gewünschten" oder „erlaubten" Systemläufe gesammelt sind. Eine „Lösung" ist ein Algorithmus, der aus den bisherigen Inputs $e_0, e_1, \ldots, e_i$ jeweils die nächste Ausgabe a_i produziert; er berechnet also eine Funktion $F: E^+ \rightarrow A$. Er ist eine Problemlösung, wenn jeder so erzeugte Systemlauf $e_0\ F(e_0)\ e_1\ F(e_0 e_1)\ e_2\ F(e_0 e_1 e_2) \ldots$ zu R gehört. Da man die jeweiligen Argumente und Werte von F wiederum durch endliche Wörter kodieren kann, lassen sich „Lösungen" wiederum durch bestehende Algorithmen (Turingmaschinen) erfassen. Jedoch ist die Problem*beschreibung* (durch eine infinitäre Relation R) komplexer als im klassischen Fall. Die hier nötige „Berechenbarkeitstheorie reaktiver Systeme" steht erst ganz am Anfang; z.B. weiß man noch nicht so gut wie in der klassischen Theorie, welche Probleme R nun „lösbar" und welche „unlösbar" sind – und erst recht fehlt hier noch eine passende Komplexitätstheorie.

-ENDE-

Klausuraufgaben

Weihnachtsaufgaben

(freiwillige Probeklausur)

Aufgabe 1 Geben Sie detailliert eine TM an, welche die Funktion $\{0,1\}^* \to \{0,1\}^*$, $x \mapsto xx$ berechnet. Kommentieren Sie die Turing-Tafel.

Aufgabe 2 Zeigen Sie, dass es zu jeder TM M eine äquivalente TM M' gibt so, dass M' niemals versucht, einen Linksschritt zu machen, wenn der Arbeitskopf auf dem linkesten Arbeitsfeld steht. (Formulieren Sie umgangssprachlich die Idee und dann die Konstruktion.)

Aufgabe 3 Wir betrachten in dieser Aufgabe $\mathtt{WHILE}_{\mathrm{mod}\ 2}$-Programme, das sind Programme, die induktiv wie $\mathtt{WHILE}_0$-Programme definiert sind, aber ohne Zuweisungen der Form $\mathtt{Xi} := \mathtt{Xi} + \mathtt{1}$ und $\mathtt{Xi} := \mathtt{Xi} \dot{-} \mathtt{1}$, dafür mit den Zuweisungen der Form $\mathtt{Xi} := \mathtt{Xi} + \mathtt{2}$ sowie $\mathtt{Xi} := \mathtt{Xi} \dot{-} \mathtt{2}$. Die semantische Funktion $[\![P]\!]$ für ein $\mathtt{WHILE}_{\mathrm{mod}\ 2}$-Programm P sei in der naheliegenden Weise definiert.
Zeigen Sie: Es gibt eine $\mathtt{WHILE}$-berechenbare Funktion, die nicht von einem $\mathtt{WHILE}_{\mathrm{mod}\ 2}$-Programm berechnet wird.
Hinweis: Induktion über den Aufbau von $\mathtt{WHILE}_{\mathrm{mod}\ 2}$-Programmen.

Aufgabe 4 Sei Σ endliches Alphabet, sei $f : \Sigma^* - \to \Sigma^*$ injektiv und berechenbar. Zeigen Sie: Die Umkehrfunktion f^{-1} ist berechenbar.

Aufgabe 5 Ein Aufzählungsverfahren, welches kein Wort mehrfach ausgibt, nennen wir *wiederholungsfrei*.
Zeigen Sie oder widerlegen Sie: Zu jeder aufzählbaren Sprache L gibt es ein wiederholungsfreies Aufzählungsverfahren, welches L aufzählt.

Aufgabe 6 Seien $m, n \geq 0$. Seien $g : \mathbb{N}^m \to \mathbb{N}$, $f_1, \ldots, f_m : \mathbb{N}^n \to \mathbb{N}$ $\mathtt{LOOP}$-berechenbare Funktionen. Die Funktion $f : \mathbb{N}^n \to \mathbb{N}$ entstehe aus $g, f_1, \ldots, f_m$ vermöge des Einsetzungsprozesses.
Zeigen Sie: f ist $\mathtt{LOOP}$-berechenbar.

Aufgabe 7 Sei $\underline{P}$ folgendes Problem:

Gegeben: TM M

Frage: Akzeptiert M das leere Wort in geradzahlig vielen Schritten?

Zeigen Sie: $\underline{P}$ ist unentscheidbar.

Aufgabe 8 Zeigen Sie: Jede konstante Funktion ist berechenbar.

Aufgabe 9 Wir betrachten Entscheidungsprobleme mit Eingabemenge $\{0,1\}^*$. Zeigen Sie: Es gibt kein Entscheidungsproblem $\underline{P} = (\{0,1\}^*, P)$ so, dass für alle Probleme $\underline{Q} = (\{0,1\}^*, Q)$ gilt: $\underline{Q} \leq \underline{P}$.
Hinweis: Betrachten Sie die Folge $\mathfrak{A}_i$ aller (Kodierungen von) Turingmaschinen (etwa in kanonischer Reihenfolge), welche totale Funktionen $f : \{0,1\}^* \to \{0,1\}^*$ berechnen. Verwenden Sie einen Diagonalschluss.

Aufgabe 10 Erinnerung: *TOTAL* ist die Menge aller Turingmachinen über $\{0,1\}^*$, welche auf allen Eingaben halten.
Sei *EVEN-TOTAL* die Menge aller Turingmachinen über $\{0,1\}^*$, welche auf allen Eingaben gerader Länge halten.
Zeigen Sie: $\underline{\textit{EVEN-TOTAL}} \leq \underline{\textit{TOTAL}}$.

Übungsschein-Klausur

Aufgabe 1 (4 Punkte) Die Funktion $f : \{0,1\}^* \to \{0,1\}^*$ sei definiert wie folgt:

$$f(a_1 \dots a_n) = \begin{cases} a_1 \dots a_n & \text{falls } n \text{ gerade,} \\ a_1 & \text{falls } n \text{ ungerade,} \end{cases}$$

wobei $a_1, \dots, a_n \in \{0,1\}$. Geben Sie detailliert eine TM an, die f berechnet.

Aufgabe 2 (4 Punkte) Sei M eine DTM, die für alle $n \geq 1$ und alle Eingaben der Länge n jeweils höchstens die ersten $2n$ Arbeitsfelder des Rechenbandes benutzt (=besucht). Geben Sie an, wie man hieraus eine DTM M' mit $\mathrm{Def}(M) = \mathrm{Def}(M')$ gewinnt (d.h. M und M' terminieren für dieselben Eingabewörter) so, dass M' für Eingaben der Länge n höchstens $n+2$ Felder benutzt.
Erläutern Sie die Arbeitsweise von M' anhand typischer Konfigurationen.
Hinweise: Es kommt auf die geeignete Wahl des Arbeitsalphabets an. Geben Sie es genau an!

Aufgabe 3 (4 Punkte) `REPEAT-UNTIL`-Programme sind analog zu den `WHILE`-Programmen definiert, wobei aber in der Syntaxbeschreibung die Einführung von `WHILE` ersetzt ist durch die Regel:

Wenn P ein `REPEAT-UNTIL`-Programm ist, so auch $\texttt{repeat}\,P\,\texttt{until}\,\texttt{Xi} = \texttt{0}$.

Entsprechend wird die semantische Funktion festgelegt: Anschaulich wird P so oft ausgeführt (mindestens jedoch einmal) bis die Variable `Xi` den Wert 0 hat.

(a) (2 Punkte) Definieren Sie die semantische Funktion von `REPEAT-UNTIL`-Programmen formal, d.h. induktiv, wobei Sie sich auf den Induktionsschritt und die Betrachtung eines `REPEAT-UNTIL`-Programms der Form

$$P_1 = \texttt{repeat}\,P\,\texttt{until}\,\texttt{Xi} = \texttt{0}$$

beschränken können. Es ist also $[\![P_1]\!](k_1, k_2, \dots)$ festzulegen.

(b) (2 Punkte) Zeigen Sie (induktiv über den Aufbau der `WHILE`-Programme), dass es zu jedem `WHILE`-Programm P ein `REPEAT-UNTIL`-Programm P' gibt mit

$$[\![P]\!] = [\![P']\!].$$

Sie können sich dabei wiederum auf den Induktionsschritt und die Betrachtung eines `WHILE`-Programms der Form $P_1 = \texttt{while Xi} > \texttt{0 begin}\, P\, \texttt{end}$ beschränken.

Aufgabe 4 (5 Punkte) Eine Sprache L heiße *monoton aufzählbar*, wenn es einen Aufzählungsalgorithmus $\mathfrak{A}$ gibt, der die Wörter in L in kanonischer Reihenfolge ausgibt.
Zeigen Sie auf intuitiver Ebene: Wenn L monoton aufzählbar ist, so ist L entscheidbar.
Hinweis: L kann endlich oder unendlich sein.

Aufgabe 5 (3 Punkte) Zeigen Sie auf intuitiver Ebene: Wenn L entscheidbar ist, so ist L monoton aufzählbar (im Sinne der Definition in Aufgabe 4).

Aufgabe 6 (5 Punkte) Ein `LOOP`-Programm heiße *if-then-else–frei*, falls darin kein `if`..`then`..`else`-Konstrukt auftaucht.
Zeigen Sie durch Induktion über den Aufbau der `LOOP`-Programme, dass es zu jedem n und jedem `LOOP`-Programm P ein if-then-else–freies `LOOP`-Programm P' gibt mit $f_P^{(n)} = f_{P'}^{(n)}$.
Sie können sich dabei auf den Induktionsschritt und die Betrachtung eines `LOOP`-Programms der Form

$$\texttt{if Xi} > \texttt{0 then begin}\, P_1\, \texttt{end else begin}\, P_2\, \texttt{end}$$

beschränken.

Aufgabe 7 (4 Punkte) Erinnerung: TOTAL (bzw. INFIN) sind die Mengen von TMen über $\{0, 1\}$, die auf allen (bzw. auf unendlich vielen) Eingaben aus $\{0, 1\}^*$ halten. Eingabemenge für die zugehörigen Entscheidungsprobleme $\underline{\text{TOTAL}}$ und $\underline{\text{INFIN}}$ sei jeweils die Menge aller TM über $\{0, 1\}$.
Zeigen Sie: $\underline{\text{TOTAL}} \le \underline{\text{INFIN}}$.

Aufgabe 8 (4 Punkte) Sei $\underline{\text{COLOR}(3)}$ das 3-Färbbarkeits-Problem für ungerichtete Graphen. Sei analog $\underline{\text{COLOR}(4)}$ das 4-Färbbarkeits-Problem für ungerichtete Graphen.
Zeigen Sie: $\underline{\text{COLOR}(3)} \le_p \underline{\text{COLOR}(4)}$.
Hinweis: Es genügt die Angabe der Reduktion (ohne expliziten Nachweis der Polynomzeitberechenbarkeit.)

Aufgabe 9 (2 Punkte) Seien $L_1, L_2 \subseteq \Sigma^*$ Sprachen in P, wobei L_2 weder $\emptyset$ noch Σ^* sei.
Zeigen Sie: $L_1 \le_p L_2$.

Aufgabe 10 (4 Punkte) Erinnerung: Für eine Sprache $L \subseteq \Sigma^*$ ist $L^* = \{w_1 \dots w_n \mid n \ge 0,\, w_1, \dots, w_n \in L\}$. (Ein Wort w gehört zu L^*, wenn es in der Form $w = w_1 \dots w_n$ mit $n \ge 0$ und $w_1, \dots, w_n \in L$ zerlegbar ist.)
Zeigen Sie: Wenn $L \in \text{NP}$, so $L^* \in \text{NP}$.

Aufgabe 11 (3 Punkte) Sei co-PRIMES die Menge aller Binärdarstellungen von natürlichen Zahlen, die keine Primzahl sind.
Zeigen Sie: $\underline{\text{co-PRIMES}}$ ist in NP.
Hinweis: Sie könnnen auf die Definition von NP zurückgehen oder aber eine polynomzeitbeschränkte NTM angeben. Sie brauchen die Polynomzeitschranke selbst nicht im Detail anzugeben und zu überprüfen.

Wiederholungs-Klausur

Aufgabe 1 (4 Punkte) Die Funktion $f : \{0,1\}^* \to \{0,1\}^*$ sei definiert wie folgt:

$$f(a_1 \dots a_n) = \begin{cases} a_{n-1} & \text{falls } n \geq 2 \\ \perp & \text{sonst} \end{cases}$$

wobei $a_1, \dots, a_n \in \{0,1\}$. Geben Sie detailliert eine TM an, die f berechnet.

Aufgabe 2 (4 Punkte) Eine *Quadrupel-TM* $(Q, \Sigma, \Gamma, q_0, q_s, \delta)$ ist ähnlich definiert wie eine TM, mit der Ausnahme, dass die Übergangsfunktion δ eine Funktion

$$(Q \setminus \{q_s\}) \times \Gamma \to Q \times (\Gamma \cup \{L, R, N\})$$

ist.
Anschaulich darf eine Quadrupel-TM in jedem Schritt *entweder* schreiben *oder* den Kopf bewegen. Formal definiert man die Bedeutung von $\delta(q,a) = (q', X)$ (d.h. der Befehlszeile (q, a, q', X)) wie folgt:

- Falls $X = L$ oder $X = R$ oder $X = N$, so wie die Bedeutung von $\delta(q,a) = (q', a, X)$ bei üblichen TM,
- Falls $X \in \Gamma$, so wie die Bedeutung von $\delta(q,a) = (q', X, N)$ bei üblichen TM.

Zeigen Sie: Zu jeder TM gibt es eine Quadrupel-TM, die dieselbe Funktion berechnet. Geben Sie die entsprechende Konstruktion genau an und begründen Sie die Richtigkeit.

Aufgabe 3 (4 Punkte) `REPEAT-UNTIL`-Programme sind analog zu den `WHILE`-Programmen definiert, wobei aber in der Syntaxbeschreibung die Einführung von `WHILE` ersetzt ist durch die Regel:

Wenn P ein `REPEAT-UNTIL`-Programm ist, so auch `repeat` P `until Xi` $= 0$.

Entsprechend wird die semantische Funktion festgelegt: Anschaulich wird P so oft ausgeführt (mindestens jedoch einmal) bis die Variable `Xi` den Wert 0 hat.
Zeigen Sie: Zu jedem `REPEAT-UNTIL`-Programm gibt es ein `WHILE`-Programm, welches dieselbe Funktion berechnet.
Hinweis: Geben Sie eine geeignete Induktionsbehauptung für alle `REPEAT-UNTIL`-Programme P an, welche sich auf die semantische Funktion $[\![P]\!]$ bezieht. Sie brauchen von dem Induktionsbeweis nur den Schritt vorzuführen, der das `repeat` ... `until` ...-Konstrukt betrifft.

Aufgabe 4 (4 Punkte) Wir betrachten in dieser Aufgabe $\texttt{WHILE}_{\text{mod }2}$-Programme, das sind Programme, die induktiv wie $\texttt{WHILE}_0$-Programme definiert sind, aber ohne Zuweisungen der Form $\texttt{Xi} := \texttt{Xi} + 1$ und $\texttt{Xi} := \texttt{Xi} \dot{-} 1$, dafür mit den Zuweisungen der Form $\texttt{Xi} := \texttt{Xi} + 2$ sowie $\texttt{Xi} := \texttt{Xi} \dot{-} 2$. (Andere Wertzuweisungen treten nicht auf.) Die semantische Funktion $[\![P]\!]$ für ein $\texttt{WHILE}_{\text{mod }2}$-Programm P sei in der naheliegenden Weise definiert.
Zeigen Sie: Es gibt eine `WHILE`-berechenbare Funktion, die nicht $\texttt{WHILE}_{\text{mod }2}$-berechenbar ist.
Hinweis: Zeigen Sie zunächst per Induktion über der Aufbau von $\texttt{WHILE}_{\text{mod }2}$-Programmen: Für alle $\texttt{WHILE}_{\text{mod }2}$-Programme P gilt:

$$\forall k_1, k_2, \dots\ l_1, l_2, \dots : \quad \text{Wenn } [\![P]\!](k_1, k_2, \dots) = (l_1, l_2, \dots) \text{ und } k_1, k_2, \dots \text{ gerade, so sind auch } l_1, l_2, \dots \text{ gerade.}$$

Aufgabe 5 (4 Punkte) Sei $L \subseteq \{0,1\}^*$ nichtleer und aufzählbar. Zeigen Sie (auf intuitiver Ebene): Es gibt eine totale, berechenbare Funktion $f : \{|\}^* \to \{0,1\}^*$ mit $\text{Bild}(f) = L$.

Aufgabe 6 (3 Punkte) Erinnerung: Das Halteproblem $\underline{H}$ lautet: „Gegeben: Eine TM M. Frage: Hält M, angesetzt auf das leere Wort?".
Das Leerheitsproblem $\underline{L}$ lautet: „Gegeben: Eine TM M. Frage: Gibt es ein Wort w so, dass M angesetzt auf w hält?"
Zeigen Sie, dass das Halteproblem auf das Leerheitsproblem reduzierbar ist.
(Formal: $\underline{H} \leq \underline{L}$.)

Aufgabe 7 (4 Punkte) Erinnerung: $\underline{\text{TOTAL}}$ ist folgendes Problem: „Gegeben: Eine TM M. Frage: Hält M auf allen Eingaben?".
Sei $\underline{\text{TOTAL8}}$ das Problem: „Gegeben: Eine TM M. Frage: Hält M auf allen Eingaben der Länge ≥ 8 ?".
Zeigen Sie: $\underline{\text{TOTAL}} \leq \underline{\text{TOTAL8}}$.

Aufgabe 8 (5 Punkte) Seien L_1, L_2 nichtleere Sprachen über dem Alphabet $\{0,1\}$. Sei \$ ein neues Symbol. Wir schreiben $L_1\$L_2$ für die Sprache $\{u\$v \mid u \in L_1,\, v \in L_2\}$.
Zeigen Sie:

(a) $L_1 \leq_p L_1\$L_2$ und $L_2 \leq_p L_1\$L_2$.

(b) Für alle Sprachen Q mit $L_1 \leq_p Q$ und $L_2 \leq_p Q$ gilt $L_1\$L_2 \leq_p Q$.

Aufgabe 9 (3 Punkte) Sei $L \subseteq \Sigma^*$ eine Sprache. Ein Wort w aus Σ^* heißt *L-Potenz*, falls es ein $n \geq 1$ und ein $u \in L$ gibt mit

$$w = \underbrace{u \ldots u}_{n \text{ mal}}.$$

Sei $\text{pot}(L)$ die Menge aller L-Potenzen.
Zeigen Sie: Falls $L \in \text{NP}$, so auch $\text{pot}(L) \in \text{NP}$.
Hinweis: Beschreiben Sie die Arbeitsweise einer entsprechenden Maschine. Sie brauchen die Laufzeitschranke nicht im Detail zu überprüfen.

Aufgabe 10 (4 Punkte) Erinnerung: $\underline{\text{CLIQUE}}$ ist folgendes Problem: „Gegeben ein Graph G und eine Zahl $n \geq 1$. Frage: Gibt es in G eine Clique der Größe n?".
Sei $\underline{P}$ folgendes Problem:

Gegeben: Graph G und Zahl $n \geq 1$.

Frage: Gibt es n verschiedene Knoten in G, die paarweise nicht durch Kanten verbunden sind?

Zeigen Sie: $\underline{P} \leq_p \underline{\text{CLIQUE}}$.

Literatur

Einführende Bücher

- W.S. Brainerd, L.H. Landweber: *Theory of Computation.* Wiley, 1974
  (Berechenbarkeitstheorie über Wörtern, ausführlich und umfassend)

- D. Harel: *Algorithmics - The Spirit of Computing.* Addison-Wesley, 1987
  (sehr lesenswerte, informelle Einführung!)

- H.R. Lewis, C. Padadimitriou: *Elements of the Theory of Computation.* Prentice-Hall, 1981
  (gut lesbares Lehrbuch zur Theoretischen Informatik)

- Z. Manna: *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974
  (enthält auch Ergebnisse zur Theorie der Programmierung)

- R. McNaughton: *Elementary Computability, Formal Languages and Automata.* Prentice-Hall, 1982
  (gut motivierende, ausführliche Einführung)

- A. Salomaa: *Computation and Automata.* Cambridge Univrsity Press, 1985
  (kompakte Darstellung)

- U. Schöning: *Theoretische Informatik – kurzgefasst.* Spektrum Akademischer Verlag, 1997
  (knapp gehaltenes Skript)

- M. Sipser: *Introduction to the Theory of Computation.* PWS Publ. Comp. 1997
  (sehr gut verständlich geschrieben, auch mit Einführungen in viele aktuelle Forschungsgebiete)

- K.W. Wagner: *Theoretische Informatik.* Springer-Verlag, 1994
  (sehr sorgfältige Darstellung der elementaren Berechenbarkeitstheorie)

- D. Wood: *Theory of Computation.* Harper & Row, 1987
  (ein „Course Book“ mit vielen Beispielen)

Weiterführende Bücher

- E. Börger: *Berechenbarkeit, Komplexität, Logik.* Vieweg, 1985
  (ein reichhaltiges Werk mit vielen Literaturhinweisen)
- M.R. Garey, D.S. Johnson: *Computers and Intractability.* Freeman, 1979
  (die „Bibel der NP-Vollständigkeit")
- P. Odifreddi: *Classical Recursion Theory.* North-Holland, 1989
  (übersichtliches, umfassendes Werk mit vielen Literaturhinweisen)
- C. Papadimitriou: *Computational Complexity.* Addison-Wesley, 1994
  (das derzeitige Standardwerk zur Komplexitätstheorie)

Index